SYSTEMATIC TESTING FOR ACTOR PROGRAMS

BY

STEVEN THOMAS LAUTERBURG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Assistant Professor Darko Marinov, Chair and Director of Research
Professor Gul Agha
Associate Professor Mahesh Viswanathan
Associate Research Professor Ralph Johnson

Abstract

The growing use of multicore and networked computing systems is increasing the importance of developing reliable parallel and distributed code. Testing such code is notoriously difficult, especially for shared-memory models of programming. The actor model of programming offers a promising alternative for developing concurrent systems based on message passing. In actor-based systems, shared-memory access is not allowed, and the key source of non-determinism is the order in which messages are delivered to and processed by the actors. As a result, errors may occur in actor programs due to the incorrect interleaving of messages, conflicting constraints on what messages can be delivered, or errors in the sequential code within individual actors. The research community has expended a great deal of effort on the testing and verification of concurrent systems. However, much of this effort has been general in nature or focused on shared-memory models.

Given the differences in how errors manifest in actor programs, it seems natural that tools and techniques for testing such programs should take into account the particular nature of the actor model. To effectively and efficiently automate the detection of these errors, we propose a set of tools and techniques specifically designed to systematically explore the different behaviors of actor programs resulting from possible message delivery schedules.

Specifically, this dissertation presents Basset, a general framework for the systematic testing of actor systems developed with languages that compile to Java bytecode. This framework provides a common set of capabilities designed and implemented to take advantage of actor semantics. By building these capabilities into a language-independent core, they are available for use by any instantiation of the framework. This dissertation illustrates the practicality and effectiveness of this approach by presenting two tool instantiations of the Basset framework: one for the Scala programming language and the other for the ActorFoundry library for Java. The implementation of Basset was built as an extension to Java PathFinder, a popular model checker for Java, in order

to leverage capabilities that already exist in that model checker. The Basset framework approach directly enables the relatively quick development of testing environments for actor-based languages and/or libraries that compile to Java bytecode.

This dissertation also considers the effectiveness of dynamic partial-order reduction techniques as they relate to the exploration of actor programs. The use of dynamic partial-order reduction speeds up systematic testing by pruning parts of the exploration space. However, the level of potential pruning is highly dependent on the order in which messages are considered for processing. This dissertation presents an evaluation of a number of heuristics for choosing the order in which messages are explored in Basset. The experiments show that the choice of heuristic can affect the number of execution paths that need to be explored by over two orders of magnitude.

Acknowledgments

This work would not have been possible without the support of many people. First of all, I would like to thank my advisor, Darko Marinov, who offered encouragement and kept me moving forward over the last several years. I also would like to specially thank the other members of my Ph.D. committee who provided valuable support and feedback throughout this effort: Gul Agha, Ralph Johnson, and Mahesh Viswanathan.

I would particularly like to thank Mirco Dotta for his work on the initial version of Basset, especially the Scala instantiation, and Rajesh Karmani for all his help developing and evaluating dynamic partial-order reduction techniques for actor programs.

I would like to thank my colleagues in the department: Holly Bagwell, Rob Bocchino, Roy Campbell, Marcelo d'Amorim, Brett Daniel, Danny Dig, Chucky Ellison, Margaret Fleck, Tom Gambill, Kely Garcia, Milos Gligoric, Mark Hills, Vilas Jagannath, Mike Katelman, Nitish Korula, Yun Young Lee, Adrian Nistor, Madhu Parthasarathy, Grigore Rosu, Ralf Sasse, Traian Serbanuta, Ahmed Sobeih, Samira Tasharofi, Elaine Wilson, and Marsha Woodbury. They were always there for me when I needed a paper reviewed, an idea discussed, or some other form of support. No doubt there are people missing from this list. I apologize to those I may have omitted.

Thanks also to the students from the Fall 2008 Concurrent Programming Languages and Fall 2009 Software Engineering classes at the University of Illinois for their feedback on this work. Additionally, I am grateful for financial support from Darko Marinov's NSF Grants, Nos. CCF-0916893 and CNS-0615372.

Finally, I would like to thank Nora, Jennie, Karen, Kevin, Bill, Mike, Agni, Sandra, Phil, Lisa, Rebecca, and Carter for their love and support over the years. I could not have done this without the help of my family and close friends.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
CLR	Common Language Runtime
DPOR	Dynamic Partial-Order Reduction
JNI	Java Native Interface
JPF	Java PathFinder
JVM	Java Virtual Machine
MJI	Model Java Interface
POR	Partial-Order Reduction
RPC	Remote Procedure Call

Chapter 1

Introduction

The growing use of multicore and networked computing systems is increasing the importance of developing reliable parallel and distributed code. Unfortunately, developing and testing such code is very hard, especially using shared-memory models of programming, which often results in concurrency bugs such as atomicity violations, data races and deadlocks. An alternative for writing parallel and distributed code is message passing, where multiple autonomous agents communicate by exchanging messages.

In the *Actor programming model*, these autonomous agents are called actors. The actor model provides message passing with object-style data encapsulation [Agh86, AMST97]. By default, messages in the actor model are *asynchronous*; other forms of communication, such as remote-procedure-call style synchronous messages, are defined in terms of asynchronous messages [Agh86]. Each actor has its own thread of control, a unique *actor name* (its virtual address), and a mailbox. If an actor is busy, messages sent to it are queued in its mailbox. When an actor is done with processing a message, it checks its mail queue for another message. In response to processing a message, an actor may do one or more of three actions: update its own local state (including updates that can change the actor's behavior), send messages to actors that it knows about, and create new actors.

Many actor-oriented programming systems have been developed, including ActorFoundry [AF], Akka [Akk], Axum [Micb], Charm++ [KK93], E [Mil06], Erlang [Arm07], Jetlang [Jet], Jsasb [Jsa], Kilim [SM08], Newspeak [New], Ptolemy [EJL+03], Revactor [Rev], Salsa [VA01], Scala [OSV08], Singularity [HL07], ThAL [Kim97], and the Asynchronous Agents Framework used in Microsoft Visual Studio 2010 image processing software [Mica]. While some of these systems are entirely new languages designed around actors, others are libraries and frameworks built for existing languages.

Taking into account the wide-spread use of the Java language, it should come as no surprise

that several of the above languages and libraries target the creation of actor programs that compile to Java bytecode. These systems include ActorFoundry, Akka, Jetlang, Jsasb, Kilim, SALSA, and Scala. The research represented in this dissertation focuses on the use of two of these systems: the Scala programming language and the ActorFoundry library for Java.

ActorFoundry [AF] is a Java framework developed at the UIUC's Open Systems Laboratory. It provides an Application Programming Interface (API) that allows developers to develop actor programs in a familiar language and includes a runtime architecture to execute those programs in an efficient manner. Programs consist of actor objects that encapsulate an actor's state and behavior, and that implicitly handle the receipt of messages via methods written by the developer. The API supports the creation of new actors, the sending of asynchronous messages to other actors, and synchronized messaging in a Remote Procedure Call (RPC) style. By default, a deep copy of message contents is made to ensure there is no shared-memory interaction between actors. ActorFoundry also supports behavior changes by providing a mechanism based on Java annotations for constraining which message types can be received. At execution time, actor objects are associated with Java threads utilizing an efficient continuation-based thread pool approach based on Kilim [SM08]. Section 2.2 provides additional details on ActorFoundry.

Scala [OSV08] is a programming language developed by Martin Odersky and his group at the EPFL's Programming Methods Laboratory. The language is a statically-typed blend of object-oriented and functional programming features. It compiles to Java bytecode and allows easy integration of Scala and Java code. The language is becoming increasingly popular and is already used for systems such as Twitter's core message queue [Ven09]. Although developers are able to utilize Java's thread-based concurrency mechanisms, the primary construct for concurrency in Scala is actors [Sca, OSV08]. Like ActorFoundry, Scala also provides classes to implement actor objects and utilizes a thread pool at execution time to provide more efficient execution [HO07]. Unlike ActorFoundry, Scala supports explicit message receipt and utilizes a pattern matching mechanism to constrain which messages can be received at a given point.

The rest of this chapter is organized as follows. Section 1.1 presents challenges in testing actor programs. Section 1.2 briefly summarizes the main statement of our work. Section 1.3 provides an overview of our actor testing framework. Section 1.4 describes our current work on using dynamic

partial-order reduction heuristics to allow more efficient exploration of actor programs. Section 1.5 summarizes the contributions of this dissertation. Finally, Section 1.6 presents an overview of the rest of this dissertation.

1.1 Challenges for Testing Actor Programs

While the research community has expended a great deal of effort on the testing and verification of concurrent systems, much of this effort has been general in nature or focused on shared-memory models. In the theoretical model of actor-based systems, however, communication via shared memory is not allowed. The key source of non-determinism is instead the order in which messages are delivered to and processed by the actors. While actor programming avoids some of the bugs inherent in shared-memory programming, e.g., low-level data races involving access to shared data, actor programs still can have bugs: for example, there may be an unsafe ordering of messages to an actor, conflicting constraints on what messages can be delivered, or incorrect sequential code within individual actors.

The *systematic testing* of an actor system requires that we exhaustively test the different orders in which messages are delivered to the actors that make up the system. Effectively, systematic testing is a form of *model checking* where the model being considered is an implementation model (i.e., the actual code that implements the system) instead of a model of the system design or an abstraction of the system, as would be used in the more traditional form of model checking. Tools which systematically explore real code have become increasingly prevalent and include BogorVM [RDH03], CMC [MPC+02], CHESS [MQ07], Java PathFinder [VHB+03], JNuke [ASB+04], and Zing [AQR+04].

Although exploring the different message orderings for an actor program is somewhat different than exploring thread interleavings, systematic testing and model checking remain appropriate techniques for checking actor program reliability by automatically finding potential concurrency bugs. Unfortunately, the use of these techniques on real actor programs is not without its own problems. The systematic testing of actor programs can be complicated by the underlying implementation of an actor system. Actor libraries often include a complex, multithreaded runtime system for execution of actor programs. While these runtime systems enable efficient *execution* of

actor programs, because of the complexity and scheduling choices in the runtime system, they can make *exploration* of such programs impossible or inefficient. The complexity of an actor system's underlying architecture interferes with our ability to test and explore what is really important: the functionality and interaction of the actors themselves. Indeed, the goal of checking actor systems is most often to *check the actor programs* and not the underlying library used to implement these programs. Given the differences in how errors manifest in actor programs, it seems natural that tools and techniques for testing such programs should take into account the particular nature of the actor model.

To address the performance issues discussed above and to take into account the nature of specific actor or distributed systems implementations, research has mostly focused on tools tailored to just a single system [AE01, AG06, BB07, FS07, SA06, YCW+09, YKKK09]. Unfortunately, while this approach enables systematic actor program testing for a particular library or language, it does not leverage similarities across different actor systems and offers little in the way of reusable components and techniques. As a result, implementations of complex algorithms must be reimplemented for each actor system.

1.2 Thesis

This dissertation presents our efforts to help alleviate some of the above listed problems on testing actor programs. Specifically, we propose a general framework and environment for the systematic testing of actor programs that compile to Java bytecode. Our thesis is that:

> *It is possible to build a general framework that (1) allows efficient exploration of actor programs written in languages that compile to Java bytecode and (2) facilitates the reuse of testing capabilities across such languages.*

In support of this thesis, this dissertation presents research in two main areas: (1) the development of a common framework along with two instantiations of the framework for ActorFoundry and Scala, and (2) the expansion and improvement of the capabilities of the framework, in particular, the use of dynamic partial-order reduction heuristics to allow more efficient systematic exploration of actor programs being explored.

1.3 Systematic Testing Framework

In this section we briefly summarize our framework for systematic testing of actor programs, which we have named Basset. This initial portion of our work supports our hypothesis that a common framework for systematic testing of actor programs can be used to provide efficient exploration of such programs, provide a means of quickly developing testing environments for different actor libraries and languages, and facilitate reuse across those multiple tools.

Basset is focused on the automated systematic testing of actor programs which have been compiled to Java bytecode. Our choice of Java reflects the popularity of this language and the availability of Java-based actor systems. Such actor programs can be written using several languages and libraries that include ActorFoundry [AF,KSA09], Akka [Akk], Jetlang [Jet], Jsasb [Jsa], Kilim [SM08], SALSA [VA01], and Scala [OSV08].

The Basset framework provides a common set of testing and state-space exploration capabilities specifically designed and implemented to take advantage of actor semantics and it facilitates their *reuse* across any instantiation of the framework. Furthermore, the framework directly enables the relatively quick *development of testing environments* for actor-based languages and/or libraries that compile to Java bytecode. Basset is able to support different actor systems with only a thin adaptation layer required for each system. Once a language adapter for a particular actor library has been created, the resulting instantiation (i.e., tool) is able to take advantage of common capabilities such as deadlock detection, state pruning, dynamic partial-order reductions, etc. We have instantiated the Basset framework for two actor languages. Specifically, we support actor programs written using the ActorFoundry [AF] library for Java and those written in the Scala language [OSV08] (which compiles to Java bytecode). Our support for Scala is based on the actor library from the standard Scala distribution.

The framework has been designed and implemented with extensibility in mind and is intended to serve as a platform for further research into the testing of actor programs. We expect that this research will directly support and facilitate additional work in this area. In fact, Bokor et al. have already used our Basset framework for their work on model checking fault-tolerant distributed protocols [BKSS11].

In order to leverage work in model checking, we built Basset on top of Java PathFinder (JPF),

a popular explicit-state model checker for Java bytecode [VHB+03, JPF]. JPF was developed at NASA for checking programs written directly in the Java language. It has been used in numerous research projects (e.g., see [JPF]). JPF provides a specialized Java Virtual Machine that supports state backtracking and control over nondeterministic choices such as thread scheduling. Prior to our work, JPF did not have any direct support for actors, i.e., for high-level choices such as message scheduling. Note that although we use JPF, our techniques could be used in conjunction with other explicit-state model checkers such as Bandera [CDH+00], BogorVM [RDH03], CMC [MPC+02], JCAT [DIS99], JNuke [ASB+04], SpecExplorer [VCST05], or Zing [AQR+04].

One might ask why it is necessary to build a new framework instead of using JPF to directly check programs written against actor libraries such as Scala and ActorFoundry. The answer lies in how these actor libraries work. The Scala and ActorFoundry libraries include complex, multi-threaded runtime systems for execution of actor programs. While these runtime systems enable efficient *execution* of actor programs, because of the complexity and scheduling choices in the runtime systems, they make *exploration* of the programs impossible or inefficient. For instance, JPF cannot even execute all the ActorFoundry library, as the ActorFoundry uses Java networking libraries for exchanging messages between actors [AG06, BB07]. JPF can execute the Scala library, but the resulting state space is huge: for example, exploration of the states of a simple Scala `helloworld` application did not complete in an hour! Even after we simplified parts of the Scala library, JPF still took *over 7 minutes* to check `helloworld`. The exploration state space is large, not because of the complexity of the actor program code, but because of the runtime architecture.

A design goal for Basset was the *efficient exploration of actor application code itself and not the exploration of the actor libraries.* Therefore, Basset does not check the actual library code but focuses instead on exposing potential bugs in the application code due to message scheduling, which is the source of non-determinism in actor-based programs. Specifically, the adaptation layer in Basset replaces the implementation of an actor library with much simpler code that still provides the same interface to the actor application but enables a much faster exploration. The result is a highly efficient system to test actor code: Basset takes *less than one second* to check `helloworld`.

Figure 1.1 provides a high-level view of the software layers that make up the Basset environment. Basset supports direct exploration of an *actor program*'s unmodified application code itself,

Figure 1.1: Basset systematic testing environment software layers

not the actor libraries and runtime architectures. A Basset instantiation for a particular language effectively replaces the language's complex runtime architecture. A *language adapter* layer developed specifically for a particular language redirects calls from the application code to the *Basset core* testing architecture. The Basset core, in conjunction with the Java PathFinder tool which it extends, performs the duties normally handled by an actor language's runtime architecture and manages the systematic exploration of different message delivery orderings as well. It is important to note that this architectural approach of using a shared, common core facilitates reuse of testing capabilities (e.g., dynamic partial-order reductions) across multiple actor languages and libraries.

To validate the implementation of the Basset framework and our tool instantiations for ActorFoundry and Scala, we ran a series of experiments using nine subjects programs, representing various communication patterns. Both ActorFoundry and Scala versions of the programs were created. The number of possible message delivery schedules needed to explore the behavior of these programs ranged from 6 for a simple client-server subject to over 60,000 for our porting of a publicly available MPI example [Pi], which computes an approximation of π by distributing it across a set of worker actors. The results for state space pruning experiments using state compar-

ison demonstrated that using an actor-specific state abstraction for comparison could reduce the number of states explored by up to 71.65% over JPF's built in state comparison and reduce the number of execution paths that need to be run in their entirety by over 83.33%.

Of particular note were the results of our experiments applying dynamic partial-order reduction (DPOR) techniques to the explorations. As expected, the pruning that resulted was quite large (up to 99.96% fewer states). However, we also noted a large variation (over two orders of magnitude) in the number states that could be pruned based on on the order in which messages were considered for processing by the DPOR algorithm. This observation is what motivated our investigation of ordering heuristics introduced in the next section. Chapter 3 provides more detail on Basset and its capabilities.

1.4 Dynamic Partial-order Reduction Heuristics

We next introduce our work on the use of heuristics to increase the efficiency of dynamic partial-order reduction (DPOR) techniques. A key challenge in testing actor programs is their inherent nondeterminism: even for the same input, an actor program may produce different results based on the *schedule* of arrival of messages. Systematic exploration of possible message arrival schedules is required both for testing and for model checking concurrent programs [CGP99, SA06, FS07, AE01, AG06, BB07, YCW+09, YKKK09, FG05, God96]. However, the large number of possible message schedules often limits how many schedules can be explored in practice. Fortunately, such exploration need not enumerate all possible schedules to check the results. *Partial-order reduction (POR)* techniques speed up exploration by pruning some message schedules that are equivalent [CGP99, SA06, YCW+09, FG05, God96, KWG09, JSM+09]. *Dynamic partial-order reduction (DPOR)* techniques [SA06, FG05, VGK08] discover the equivalence dynamically, during the exploration of the program, rather than statically, by analyzing the program code. The actual dynamic executions provide more precise information than a static analysis that needs to soundly over-approximate a set of feasible executions. Effectively, based on the exploration of some message schedules, a DPOR technique may find that it need not explore some other schedules. This allows DPOR techniques to prune a substantial part of the exploration space.

It turns out that pruning using DPOR techniques is highly sensitive to the order in which

messages are considered for exploration. For example, consider a program which reaches a state where two messages, m_1 and m_2, can be delivered to some actors. If a DPOR technique first explores the possible schedules after delivering m_1, it could find that it need not explore the schedules that first deliver m_2. But, if the same DPOR technique first delivers m_2, it could happen that it cannot prune the schedules from m_1 and thus needs to perform the entire exhaustive exploration. We recently observed this sensitivity in our work on building a systematic testing framework for actors described in the previous section, and Godefroid mentioned it years ago [God96]. Dwyer et al. [DPE06] consider the use of heuristics in determining search paths that will more quickly find errors in shared-memory systems. However, we are not aware of any prior attempt to analyze what sorts of message selection orders lead to better pruning in DPOR for message-passing systems.

We consider the following questions regarding message-ordering heuristics:

- What are some of the natural *heuristics* for ordering scheduling decisions in DPOR for message-passing systems?
- What is the impact of choosing one heuristic over another heuristic?
- Does the impact of these heuristics depend on the DPOR technique?
- Can we predict which heuristic may work better for a particular DPOR technique or subject program?

We define eight different heuristics and evaluate the impact each has on pruning the search space for several test subjects. The evaluation is performed using three different DPOR implementations: one based on the algorithm used for dCUTE [SA06] (see Section 2.7.2) and the other two based on dynamically computing persistent sets [FG05, God96] (see Section 2.7.1). The persistent set technique was considered both stand-alone and in combination with sleep sets [God91, God96]. As our evaluation platform, we use the Basset framework described in Section 1.3.

The evaluation results show that different heuristics can lead to substantial differences in pruning, up to two orders of magnitude. In Chapter 4 we summarize the advantages and disadvantages of various heuristics. In particular, we point out what types of programs, based on the communication pattern of the actors, may benefit the most from which heuristics. For example, in pipelined computations, it is more efficient to first schedule messages for the early stages in the pipeline.

These insights provide important *guidelines* for exploring actor programs in practice: based on the type of the program, the user can instruct an exploration tool to use a heuristic that provides better pruning, resulting in a faster exploration and more efficient bug finding.

1.5 Contributions

This dissertation makes the following contributions:

- **I**ntroduces the concept of a general framework for exploration of actor programs that explicitly takes into account the nature of these programs.

- **P**rovides an implementation of the general framework concept in a tool called Basset which uses the Java PathFinder model checker. Basset has been released as a publicly available extension for JPF called jpf-actor. Basset can be downloaded from either the NASA JPF website [JPF] or the Basset homepage [Bas].

- **P**rovides instantiations of the Basset framework for actor programs written in the ActorFoundry library and in the Scala programming language. These instantiations illustrate the viability of and value provided by the framework concept in general and the Basset framework in particular. The instantiations also provide the first state exploration engines for these two actor systems, which are based on very different design decisions.

- **I**ncorporates two known optimization techniques: dynamic partial-order reduction [SA06, CGP99] and state comparison/hashing [SL08, CGP99] in Basset. Due to the nature of the Basset framework, these two techniques for speeding up exploration were automatically available for use with both the ActorFoundry and the Scala instantiations of the Basset framework, thus illustrating the reuse of tools and techniques made possible by the common framework concept.

- **E**valuates the Basset framework on several subjects. The evaluation shows that a single framework can systematically explore programs in an effective manner for multiple languages. Additionally, we show that Basset's approach of exploring only application code (and not un-

derlying system libraries) allows for more efficient exploration than if the underlying runtime architecture of the actor system were also considered.

- **I**dentifies and presents eight ordering heuristics that can be applied when using dynamic partial-order reduction to limit the search space during systematic testing of actor-based programs.

- **E**valuates these ordering heuristics for three DPOR techniques: one based on the algorithm used for dCUTE and two others based on persistent sets. The persistent set technique was considered both stand-alone and augmented with sleep sets.

- **S**ummarizes the observed advantages and disadvantages of the identified heuristics, and presents preliminary guidelines regarding the use of heuristics based on the characteristics of the program under test.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of key concepts and software used throughout the dissertation. This includes a discussion of the actor programming model, ActorFoundry library, Scala programming language, Java PathFinder framework and dynamic partial-order reduction. Chapter 3 offers a more detailed look at the Basset systematic testing framework and its ActorFoundry and Scala instantiations. This chapter includes both an illustrative example and experimental results. Chapter 4 follows with a presentation of our work on message-ordering heuristics for improving the efficiency of dynamic partial-order reductions. Finally, Chapter 5 discusses work related to the material presented in the dissertation, and Chapter 6 concludes with a summarization of the contributions of our work and a brief discussion of possible future work related to this research.

Chapter 2

Background

In this chapter we discuss concepts and software that are used throughout the remainder of this dissertation. Section 2.1 provides a brief introduction to the actor programming model. Sections 2.2 and 2.3 present overviews of the ActorFoundry library and Scala programming language, respectively. Section 2.4 introduces the concept of state-space exploration. Section 2.5 provides an overview of the Java PathFinder program model checker. Section 2.6 discusses how state-space exploration facilitates the systematic testing of actor programs. Section 2.7 discusses the role dynamic partial-order reduction plays in improving the efficiency of that state-space exploration.

2.1 Actor Programming Model

In the *actor* programming model [Agh86, AMST97], an actor is an autonomous concurrent object which interacts with other actors by explicitly sending messages. By default, these messages are *asynchronous*; other forms of communication, such as remote-procedure-call style synchronous messages, are defined in terms of asynchronous messages [Agh86]. Conceptually, each actor is viewed as having its own thread of control, a unique *actor name* (its virtual address), and a mailbox, as depicted in Figure 2.1. If an actor is busy, messages sent to it are queued in its mailbox. When an actor is done with processing a message, it checks its mail queue for another message. In response to processing a message, an actor may do one or more of three actions:

- **Send messages:** Messages may be sent to other actors or to itself, provided the sender knows the name of the recipient.
- **Create new actors:** Newly created actors have their own unique names and an associated mail queue. Upon creation, only the creator knows the name of the new actor, but names may be communicated in messages.

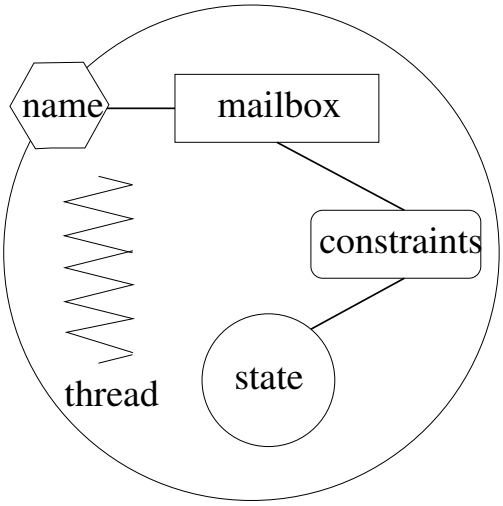

Figure 2.1: Schematic representation of an actor

- **Update local state:** Changes can be as straightforward as updates to local variables or they can change the future behavior of the actor. For example, the changes can determine the types of messages that the actor will accept in the future.

An actor program may have different executions (even for the same input) based on the interleaving of messages exchanged between the actors, in much the same way as a multithreaded program may have different executions based on the interleaving of accesses to shared memory.

Each actor operates independently. Consequently, if two actors send messages concurrently, the order of arrival of those messages is indeterminate. Moreover, no assumption is made about the routing of messages; therefore, two messages sent from the same actor to the same recipient may arrive in any order. Although in some variants of actors message order between pairs of actors is maintained, not requiring the order of messages between two actors to be maintained by default allows greater flexibility in the implementation: for example, a client may send requests to an actor operating as a common receptionist for requests. This actor then forwards it to a stateless server. If the order of messages is not required to be preserved, the behavior of the system would be the same whether the receptionist forwards the requests to a single actor, or to one of a collection of actor servers. Of course, message order can always be explicitly constrained; this requires imposing additional synchronization protocols. In some actor languages, message order between two actors can be easily expressed in terms of constraints (see below).

While the actor model does not guarantee in-order message delivery, it does guarantee that the messages are eventually delivered. Of course, in a real distributed implementation, messages may be lost, just as processors may crash or buffers may overflow in an implementation of a sequential program. Using an abstraction that guarantees message delivery enables us to reason about the liveness properties of actor systems. Lossy messages can always be modeled by explicitly representing channels as actors which nondeterministically lose messages. This is not difficult to do, but can be prohibitively expensive during the testing of reasonably complex systems due to state-space explosion. Although the behavior of an actor in response to a message is deterministic, there are a number of ways of essentially causing the behavior to be governed by a nondeterministic coin flip. The simplest is to make use of an actor which sends itself a message, causing the next message to be ignored rather than processed. Since this message would be shuffled with other messages, whether some message is lost or not would be nondeterministic.

Because of the asynchrony inherent in distributed systems, it is generally not feasible for the sender to know what the state of a recipient will be when it receives a message. For example, a free printer may ask a spooler for a print job, but the spooler may be empty. In many actor languages, the developer may specify synchronization *constraints* which are used to postpone requests until the recipient is in a state where it can process the request [HA92]. Actor runtimes implement these constraints by reordering messages. Moreover, actor languages typically provide higher level language constructs for "synchronous" or request-reply communication, where a value is returned by the actor receiving a message; the sending actor waits until this value is received before carrying out further computation. These constructs can be translated into basic actor constructs [MT97].

Actors enable programmers to think in terms of objects as agents, a natural view of concurrency. Because an actor processes only one message at a time, access to an actor's state is *serialized*. Thus, actors avoid low level data races to variables in its state, and avoid the potential for deadlocks related to preventing those races. In essence, actors raise the level of abstraction, allowing larger computation steps to be viewed as a logical unit.

Although certain types of bugs inherent in shared-memory based programs are avoided, bugs may still occur in actor code: the interleaving of messages to an actor whose ordering should have been constrained may result in an incorrect behavior, the use of synchronous communication or

synchronization constraints can result in a deadlock, or the sequential code within an actor can have errors. Some sources of bugs in actor programs are the result of the deficiencies in the current generation of actor languages and libraries. For example, a critical issue for actor programs is the cost of sending messages. In many actor languages and libraries, the cost of message passing is minimized by *transferring ownership* rather than by copying data. Unfortunately, these actor languages and libraries require the programmer to ensure that passing messages by transferring data ownership is correct: i.e., the sender does not subsequently attempt to access the data. This dissertation does not deal directly with this particular problem. The tools and techniques presented herein assume that data is either copied or that ownership issues are correctly handled.

As mentioned in the introduction, some actor-oriented programming systems include ActorFoundry [AF], Akka [Akk], Axum [Micb], Charm++ [KK93], E [Mil06], Erlang [Arm07], Jetlang [Jet], Jsasb [Jsa], Kilim [SM08], SALSA [VA01], Scala [OSV08], Newspeak [New], Ptolemy II [EJL+03], Revactor [Rev], ThAL [Kim97], Singularity [HL07], and the Asynchronous Agents Framework used in Microsoft Visual Studio 2010 image processing software [Mica]. Many of these systems support use of languages or libraries that generate executable Java bytecode [AF, Akk, Jet, Jsa, SM08, VA01, OSV08, HO07]. This involves implementing Java classes for actor names, mail queues, threads, and state. Moreover, constraints which filter messages based on the state of an actor may be specified. The commonality of these structures is a part of what motivates us to develop a unified, generic framework in which actor state exploration can be performed and optimized. Of particular note are the ActorFoundry actor library for Java [AF] and the Scala programming language [OSV08]. These two systems were used extensively in the research underlying this dissertation.

2.2 ActorFoundry

Let us consider one of the actor programming languages for which we created a testing tool instance based on our Basset framework. The imperative actor language *ActorFoundry* is implemented as a Java framework [KSA09]. Figure 2.2 shows a Hello World program in ActorFoundry that illustrates some key actor constructs.

Classes such as `HelloActor` that describe an actor's behavior extend `osl.manager.Actor`. An

```
public class HelloActor extends osl.manager.Actor {
    ActorName worldActor = null;

    @message
    public void hello() throws RemoteCodeException {
        worldActor = create(WorldActor.class);
        call(stdout, "print", "Hello ");
        send(worldActor, "world");
    }
}

public class WorldActor extends osl.manager.Actor {
    @message
    public void world() throws RemoteCodeException {
        call(stdout, "print", "Hello ");
    }
}
```

Figure 2.2: "Hello World" program in ActorFoundry illustrating several constructs

actor may have local state comprised of primitive values and objects, and this local state is assumed to *not* be shared among actors. In ActorFoundry, an actor can communicate with another actor in the program by sending asynchronous messages using the library method `send`. The sending actor does not wait for a message to arrive at the destination and be processed. Rather, it continues execution with the next program statement as soon as the message is sent. In contrast, the library method `call` sends a synchronous message to an actor. This is actually accomplished by sending an asynchronous message, but then blocking the sender until the message is delivered and processed at the receiver and a reply is returned. An actor definition includes method definitions that correspond to messages that the actor can accept and these methods are annotated with `@message`. Both `send` and `call` can take arbitrary number of arguments that correspond to the arguments of the corresponding method in the destination actor class.

The library method `create` creates an actor instance of the specified actor class. It can take arbitrary number of arguments that correspond to the arguments of the constructor. Message parameters and return types should be of the type `java.io.Serializable`. The library method `destroy` kills the actor calling the method. However, actors cannot destroy other actors. Messages sent to the killed actor are never delivered. Note that both `call` and `create` may throw a checked exception `RemoteCodeException` indicating that a failure occurred in a remote actor or that a new actor could not successfully be created.

We next present an informal semantics for relevant ActorFoundry constructs so we are able to more precisely describe the algorithms in Section 2.7. Consider an ActorFoundry program P consisting of a set of actor definitions including a *main* actor definition that receives the initial message. `send`(a, msg) appends the contents of the message msg to the message queue of actor a. Each actor has a queue; we will use Q_a to denote the message queue of an actor a. We assume that at the beginning of execution the message queue of all actors is empty.

The ActorFoundry runtime first creates an instance of the main actor and then sends the initial message to it. Each actor executes the following steps in a loop: remove a message from the queue (termed as an *implicit receive* statement from here on), decode the message, and process the message by executing the corresponding method. During the processing, an actor may update the local state, create new actors, and send more messages. An actor may also throw an exception. If its message queue is empty, the actor blocks waiting for the next message to arrive. Otherwise, the actor *nondeterministically* removes a message from its message queue. The nondeterminism in choosing the message models the asynchrony associated with message passing in actors. In other words, we can model the delivery order nondeterminism by nondeterministically choosing from among all available messages.

An actor executing a `create` statement produces a new instance of an actor. We assume that the new actor is assigned a fresh integer identifier obtained by incrementing a global counter. An actor is said to be *alive* if it has not already executed a `destroy` statement or thrown an exception. An actor is said to be *enabled* if the following two conditions hold: the actor is alive, and the actor is not blocked due to an empty message queue or executing a `call` statement.

A variable pc_a represents the program counter of the actor a. For every actor, pc_a is initialized to the implicit receive statement. Conceptually, we can describe the semantics of an actor program as being executed by a scheduler that executes one actor at a time. This scheduler executes a loop inside which it *nondeterministically* chooses an enabled actor a from the set $\mathcal{P}$. It executes the next statement of the actor a, where the next statement is obtained by calling $statement\_at(pc_a)$. During the execution of the statement, the program counter pc_a of the actor a is modified based on the various control flow statements; by default, it advances to the next statement.

The concrete execution of an internal statement, i.e., a statement not of the form `send`, `call`,

`create`, or `destroy`, takes place in the usual way for imperative statements. The loop of the scheduler terminates when there is no enabled actor in $\mathcal{P}$. The termination of the scheduler indicates either the normal termination of a program execution or a deadlock state (when at least one actor in $\mathcal{P}$ is waiting for a `call` to return).

2.3 Scala

The Scala programming language was conceived by Martin Odersky and his research group at EPFL. It is a statically-typed general purpose language that is essentially a blend of object-oriented and functional programming features. Scala primarily supports development of programs that run on the Java Virtual Machine (JVM). It also supports development for Microsoft's .NET Framework CLR, but for purposes of this dissertation we are interested only in the version which compiles to Java bytecode. Scala code can integrate quite seamlessly with Java code as they share a common runtime representation and both run on a JVM. No doubt this interoperability has been a factor in the language's increasing popularity; it is already being used for systems such as Twitter's core message queue [Ven09].

The primary mechanism for concurrency in Scala is actors [Sca, OSV08]. Like ActorFoundry, Scala also has a complex runtime architecture that manages the actors and their communication. We illustrate some of the actor-related language features of Scala using the simple Scala example in Figure 2.3. The ClientServer driver code first creates two Actor objects—a server and a client—using the `new` instruction in Scala. This creates an object just like it would for any other class and returns an object reference. The driver then starts the two actors by invoking their `start` methods.

Our example `Server` actor simply stores and retrieves an integer value. In Scala, each actor extends the `Actor` trait. Here, the `Server` actor uses `react` to explicitly request and process a message from its mailbox. Scala uses *pattern matching* to specify which messages it can receive and what action to perform. Messages that do not match any of the cases are left in the mailbox for future processing. Essentially, the program calls `react` passing in a partial function expressed as a series of cases. For our example, the `Server` actor accepts three types of messages: (1) integers which it stores in the variable `value`, (2) the string `get` which results in a call to the `reply` method where sender information is extracted from the original message and a message containing the reply

```
object ClientServer extends Application {
    val server = new Server
    val client = new Client(server)
    client.start
    server.start
}
class Client(server: Actor) extends Actor {
    var v1: Int = _
    var v2: Int = _
    def act() = {
        server ! 5
        v1 = (server !? "get").asInstanceOf[Int]
        println("value v1 = " + v1)
        v2 = (server !? "get").asInstanceOf[Int]
        println("value v2 = " + v2)
        // assert (v1 == v2);
        server ! "shutdown"
    }
}
class Server extends Actor {
    var value: Int = _
    def act() =  loop {
        react {
            case v: Int => value = v
            case "get" => reply(value)
            case "shutdown" => exit()
        }
    }
}
```

Figure 2.3: Simple Scala client/server example

contents is sent to the original sender, and (3) the string `shutdown`, which instructs this actor to terminate itself by calling the `exit` method.

Our example `Client` actor communicates with the server. The client first uses the `!` operator to asynchronously send a message containing the value 1 to the `Server` actor. The client then uses the `!?` operator to send a synchronous `get` message to the server to retrieve the current stored value. Note that the `!?` operator essentially performs a remote procedure call. In other words, after the client sends the `get` message, it blocks until it receives a reply from the server and then stores the return value into the variable `v1` or `v2`. After the client gets the value a second time, it finally terminates the server by asynchronously sending it a `shutdown` message using the `!` operator. It should be noted that this example has multiple different behaviors depending on the order in which messages are delivered to the server. A detailed discussion of these behaviors, along with an ActorFoundry version of this example, can be found in Section 3.1.

2.4 State-Space Exploration

Systematic testing and model checking can improve program reliability by automatically finding potential bugs. A key element of these approaches is *state-space exploration* (e.g., [CGP99]). To explore the complete state space of a program for a given input, one needs to explore all feasible execution paths of the program.

More specifically, state-space exploration starts from an initial program state and explores the states that could be reached by some execution of the program. Different exploration paths may occur as a result of non-deterministic choices encountered during the execution (e.g., which thread should be executed next or, for actor programs, which message should be delivered next).

2.5 Java PathFinder

Java PathFinder (JPF) is an extensible, explicit-state model checker for Java [VHB+03, JPF] developed at NASA Ames. The initial version of JPF [Hav99] was developed as a translator from Java to Promela, the language used by the SPIN model checker [Hol97]. Subsequently, it was redeveloped as a specialized Java Virtual Machine (JVM) that supports the direct exploration and systematic testing of unmodified Java programs. JPF itself is written in Java and runs on top of a host, native JVM.

Key to the tool's raison d'être, is its ability to provide control over non-deterministic choices. The default choices in JPF are thread scheduling and explicit choices made in the code with the JPF library call `Verify.getInt` (analogous to `VS_toss` in VeriSoft [God97]). To enable the exploration of the state space that results from these choices, JPF supports efficient state backtracking to restore previously visited states. The tool uses storing and restoring of states as opposed to the re-execution approach found in tools like CHESS [MQ07] or dCUTE [SA06]. JPF also supports the pruning of exploration paths, providing both standard state comparison capabilities and the option of creating custom state comparators. JPF also supports several POR techniques.

JPF cannot directly handle *native methods* (i.e., methods written in languages other than Java, such as C, C++, or assembly). To alleviate this problem, JPF provides an interface, called the Model Java Interface (MJI), for communication between the specialized JVM and the host JVM

(analogous to the Java Native Interface (JNI) used for communication between Java and C in JVMs written in C). The MJI interface is also extremely useful for both extending the capabilities of JPF and for performance purposes. For example, MJI allows applications running on JPF to access the internal information used by the interpreter. Also, since JPF is a bytecode interpreter that runs on top of a regular JVM, the execution of Java programs on JPF is significantly slower than executing them directly on a regular JVM. Performance critical components of an application can be reimplemented at the MJI level.

2.6 State-Space Exploration and Actor Programs

To systematically test an actor program for a given input, we need to explore all *distinct* execution paths of that program. In other words, it is necessary to explore the state-space of the program. In this section, we consider the *exhaustive* exploration of an actor program in more detail. In the following section, we take a look at how one might perform a more efficient exploration.

2.6.1 Naïve Exploration

An execution path can be viewed intuitively as a sequence of program statements executed, or as we will see later, it suffices to have just a sequence of messages received. In this work, we assume that the program always terminates and a test harness is available, and thus we focus on exploring the paths for a given input. In this case, a simple, systematic exploration of an actor program can be performed using a *naïve* scheduler: beginning with the initial program state, the scheduler nondeterministically picks an enabled actor and executes the next statement for that actor. If the next statement is implicit receive, the scheduler nondeterministically picks a message for the actor from its message queue. The scheduler continues to explore a path in the program by making these choices at each step. After completing execution of a path (i.e., when there are no additional statements to be executed), the scheduler backtracks to the last scheduling step (in a depth-first strategy) and explores alternate paths by picking a different enabled actor or a different message from the ones chosen previously.

2.6.2 Macro-Steps

Note that the number of paths explored by the naïve scheduler is exponential in the number of enabled actors and the number of program statements in all enabled actors. However, an exponential number of these schedules are equivalent. A crucial observation is that actors do not share state: they exchange data and synchronize only through messages. Therefore, it is sufficient to explore paths where actors interleave at message receive points only. All statements of an actor between two implicit receive statements can be executed in a single *atomic* step called a *macro-step* [SA06, AMST97]. At each step, the scheduler nondeterministically picks an enabled actor and a message from the actor's message queue. The scheduler records the ids of the actor and the message, and executes the program statements as a macro-step. A sequence of macro-steps, each identified by an actor and message pair (a, m), is termed a *macro-step schedule*. At the end of a path, the scheduler backtracks to the last macro-step and explores an alternate path by choosing a different pair of actor and message (a, m).

2.7 Dynamic Partial-Order Reduction

Note that the number of paths explored in the previous section using a macro-step scheduler is exponential in the number of deliverable messages at each step. This is because the scheduler, for every step, executes all permutations of actor and message pairs (a, m) that are enabled before the step. However, messages sent to different actors may be independent of each other, and it may be sufficient to explore all permutations of messages for a *single* actor instead of all permutations of messages for *all* actors [SA06].

An important optimization for state-space exploration is partial-order reduction (POR). POR techniques attempt to limit exploration to all *distinct*, feasible behaviors by minimizing the number of redundant execution paths that are explored [CGP99, God96]. POR achieves this using an independence relation among events to be processed [God96]. Dynamic POR (DPOR) techniques compute this independence dynamically during the program exploration [FG05, SA06, VGK08], which can make them more precise than static POR techniques that need to soundly over-approximate possible message interactions.

The independence between certain events results in equivalent paths, in which different orders of independent events occur. The equivalence relation between paths is exploited by dynamic partial-order reduction (DPOR) algorithms to speed up automatic testing of actor programs by pruning parts of the exploration space. For actor programs, one form of equivalence can be captured dynamically using the *happens-before relation* [Fid88,Lam78,SA06], which yields a partial order on the state transitions in the program. The goal of augmenting a scheduler with a DPOR algorithm is to avoid exploring redundant paths, i.e., to explore only one or a small number of linearizations for each partial order or equivalence class.

We next describe two stateless DPOR algorithms for actor programs: one based on dynamically computing persistent sets [FG05] (adapted for testing actor programs) and the other one used in dCUTE [SA06]. Note that the algorithms presented below re-execute the program from the beginning with the initial state in order to explore new program paths. However, the algorithms can easily be modified to support checkpointing and restoration of intermediate states, because these operations do not change DPOR fundamentally.

2.7.1 DPOR based on Persistent Sets

Flanagan and Godefroid [FG05] introduced a DPOR algorithm that dynamically tracks dependent transitions and computes persistent sets [God96] among concurrent processes. They presented the algorithm in the context of shared-memory programs. Figure 2.4 shows our adaptation of their algorithm for actor programs, which also incorporates the optimization discussed by Yang et al. [YCGK07]. The algorithm computes *persistent sets* in the following way.

The *scheduler* method is responsible for controlling a single execution of an actor program. The scheduler is repeatedly called to re-execute the program from the beginning with the initial state in order to explore new program paths. This is done until the *compute_next_schedule* method indicates the exploration is completed. The *scheduler* method takes as input an actor program $\mathcal{P}$ and keeps track of the following data as it executes: program counters pc_{a_1} through pc_{a_n} for each actor in the program, message queues Q_{a_1} through Q_{a_n} for each actor, and a counter i that keeps track of the current transition. The algorithm also tracks information about the scheduling points that are encountered during executions, which it maintains as a list of transitions *path_c*.

```
scheduler(P)
  pc_{a_1} = l_0^{a_1}; pc_{a_2} = l_0^{a_2}; ...; pc_{a_n} = l_0^{a_n};
  Q_{a_1} = [ ]; Q_{a_2} = [ ]; ...; Q_{a_n} = [ ];
  i = 0;
  while (∃a ∈ P such that a is enabled)
    (a,msg_id) = next(P);
    i = i + 1;
    s = statement_at(pc_a);
    execute(a, s, msg_id);
    s = statement_at(pc_a);
    while (a is alive and s ≠ receive(v))
      if s is send(b, v)
        for all k ≤ i
              such that b == path_c[k].receiver
              and canSynchronize(path_c[k].s, s)
            // actor a' "causes" s
            path_c[k].S_p.add((a', _));
      execute(a, s, msg_id);
      s = statement_at(pc_a);
  compute_next_schedule();

compute_next_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].S_p is not empty
      path_c[j].schedule =
        path_c[j].S_p.remove();
      path_c = path_c[0 ... j];
      return;
    j = j − 1;
  if (j < 0) completed = true;

next(P)
  if (i ≤ |path_c|)
    (a,msg_id) = path_c[i].schedule;
  else
    (a,msg_id) = choose(P);
    path_c[i].schedule = (a,msg_id);
    path_c[i].S_p.add((a, _));
  return (a,msg_id);
```

Figure 2.4: Dynamic partial-order reduction algorithm based on persistent sets

The scheduler consists of two while loops, one nested inside the other. The outer loop drives the overall execution of the actor program by selecting messages to be delivered to actor. As long as an actor with a deliverable message exists, the scheduler will continue to execute the program. The message to deliver is selected by calling the *next* method. During the initial run of the program, for every scheduling point, the *next* method *nondeterministically* selects an enabled actor (as represented by a call to the *choose* method, which is underlined) and adds all of the pending messages for the selected actor to the persistent set S_p. For subsequent re-executions of the program, the method returns the message that was recorded in the previous execution (modulo changes made by the *compute_next_schedule* method at the end of the previous execution). If the transition counter has advanced beyond the point that there are previously recorded transitions, then new transitions are *nondeterministically* selected.

The inner while loop processes program statements until the actor completes processing the current message or encounters a *receive* statement. In other words the loop executes all of the statements in a macro-step (as defined in Section 2.6). If the current statement is a `send`(a, v) statement, say at transition i in the current schedule, the schedule analyzes all the *receive* statements executed by a earlier in the same execution path (represented as *path_c*). If a receive, say at position $k < i$ in the schedule, is not related to the send statement by the happens-before relation (checked in the call to method *canSynchronize*), the scheduler adds pending messages for a new actor a' to the persistent set S_p at position k in *path_c*. The actor a' is "responsible" for the send statement at i, i.e., a receive for a' is enabled at k, and it is related to the send statement by the happens-before relation. The code that determines actor a' is not shown here.

When there are no deliverable messages remaining, the execution of the actor program $\mathcal{P}$ is complete. The schedule calls the *compute_next_schedule* method to determine the schedule for the next re-execution of the program. The next schedule is essentially the schedule that was just executed with some small adjustments. The final transition in *path_c* is updated so that a different message is selected upon re-execution. Specifically, *path_c.schedule* is selected from the persistent set S_p. If the persistent set is empty, there are no more choices for this transition, and the number of transitions in the path is reduced by one. If all transitions are removed from the path, no further executions are necessary (i.e., the exploration is complete).

```
scheduler(P)
  pc_{a_1} = l_0^{a_1}; pc_{a_2} = l_0^{a_2}; ...; pc_{a_n} = l_0^{a_n};
  Q_{a_1} = [ ]; Q_{a_2} = [ ]; ...; Q_{a_n} = [ ];
  i = 0;
  while (∃a ∈ P such that a is enabled)
    (a,msg_id) = next(P);
    i = i + 1;
    s = statement_at(pc_a);
    execute(a, s, msg_id);
    s = statement_at(pc_a);
    while (a is alive and s ≠ receive(v))
      if s is send(b, v)
        for all k ≤ i
              such that b == path_c[k].receiver
              and canSynchronize(path_c[k].s, s)
            path_c[k].needs_delay = true;
        execute(a, s, msg_id);
        s = statement_at(pc_a);
  compute_next_schedule();

compute_next_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].next_schedule ≠ (⊥, ⊥)
      (a, m) =path_c[j].schedule;
      (b, m') =path_c[j].next_schedule;
      if a == b or path_c[j].needs_delay
        path_c[j].schedule =
          path_c[j].next_schedule;
        if a ≠ b
          path_c[j].needs_delay = false;
        path_c = path_c[0 ... j];
        return;
    j = j − 1;
  if (j < 0) completed = true;

next(P)
  if (i ≤ |path_c|)
    (a,msg_id) = path_c[i].schedule;
  else
    (a,msg_id) = choose(P);
    path_c[i].schedule = (a,msg_id);
  path_c[i].next_schedule = next(a,msg_id);
  return (a,msg_id);
```

Figure 2.5: Dynamic partial-order reduction algorithm based on dCUTE approach

2.7.2 DPOR in dCUTE

Figure 2.5 shows the DPOR algorithm that is a part of the dCUTE approach for testing open, distributed systems [SA06]. Since we do not, in this context, consider open systems (i.e., systems that interact with external entities in their environment), we ignore the input generation aspects of dCUTE. The algorithm proceeds in a fashion quite similar to the algorithm for persistent sets: during the initial run of the program, for every scheduling point, the scheduler *nondeterministically* picks an enabled actor (call to the *choose* method, which is underlined) and explores permutations of messages enabled for the actor. During the exploration, if the scheduler encounters a send statement of the form `send`(a, v), it analyzes all the receive statements seen so far in the same path. If a receive statement is executed by a, and the send statement is not related to the receive in the happens-before relation, the scheduler sets a flag at the point of the receive statement. The flag indicates that all permutations of messages to some other actor a' (different from a) need to be explored at the particular point. The exploration proceeds in a nondeterministic fashion again from there on. A more detailed discussion of the algorithm can be found in [SA06].

It is apparent that the two algorithms presented in Figure 2.5 and Figure 2.4 are quite similar. Both initially consider a set of all messages for a single actor a at some point in the execution of the program and, while exploring the paths that begin with the receive event for one of those messages, the algorithms look for message send events to the actor a that are *not* related (by the happens-before relation) to the receive event of one the originally considered messages to a.

The key difference between the algorithms is how they add messages to that original set. The persistent set algorithm adds all messages for the actor a' whose receive event was related to the above message send event. In contrast, the dCUTE algorithm also adds all the messages for another actor, but selects that actor in a somewhat arbitrary way. The algorithm merely indicates that all messages to some other actor a' (different from a) must be considered. The selection of that actor is nondeterministic or, more accurately, implementation dependent.

Chapter 3

Systematic Testing Framework

In this chapter we present the overall structure of our systematic testing framework, which we have named Basset.[1] the capabilities provided by the framework. The core of the framework contains common, language-independent functionality for running actor programs and exploring different message schedules. It interfaces with adaptation layers that are developed for specific actor languages. As discussed previously, the goal for Basset is to provide a platform *efficient for state-space exploration* but *not necessarily efficient for straightline execution* of actor programs. This goal affects the design decisions we made for the execution of actors.

This chapter is organized as follows. We first present an example actor program to illustrate key actor concepts and our approach to exploring actor programs in Section 3.1. Then, in Section 3.2, we introduce the overall architecture of the framework. Section 3.3 presents the *actor program* layer, including a discussion of the framework's test drivers. Section 3.4 presents the responsibilities and capabilities of the *Basset core*. Section 3.5 discusses optimizations we built into Basset to facilitate more efficient exploration of actor programs. Section 3.6 describes our integration with the Java PathFinder (JPF) framework that underlies Basset. Section 3.7 introduces our framework instantiations (i.e., language adapters) at a high level. Sections 3.8 and 3.9 provide more details on the instantiations for ActorFoundry and Scala, respectively. Section 3.10 presents the results of experiments performed using Basset. Section 3.11 concludes our discussion of this work.

[1] Some of the material presented in this chapter, as well as Sections 1.3 and 2.1, is derived from previously published work [LDMA09]. The original publication is available at `http://www.computer.org/` and the copyright is held by IEEE: A Framework for State-Space Exploration of Java-Based Actor Programs. Steven Lauterburg, Mirco Dotta, Darko Marinov and Gul Agha. Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. pages 468–479, IEEE Computer Society. © 2009 IEEE.

3.1 Example

To illustrate some key actor concepts along with the Basset approach to exploration of actor programs, we use a simple example. Our example is a simplified version of a sample actor program available on ScalaWiki [ScW], a popular web site that provides several widely used resources for Scala, including code samples contributed by some developers of the Scala programming language. Using Basset, we discovered a bug in this sample code. Here, we present a version of the code implemented using Java and the ActorFoundry library, assuming that the readers are more familiar with Java than with Scala. For comparison purposes, a simplified version of the program written in Scala is presented in Figure 2.3.

Figure 3.1 shows the code for our example. The test driver first creates two actors—a server and a client—using the `create` method from ActorFoundry. This method takes the class of the actor (ActorFoundry heavily uses Java reflection) and, optionally, arguments for the class' constructor. The method creates an actor and returns an `ActorName` object that represents a handle to the actor. (Because ActorFoundry supports distributed applications and mobility of actor code, `create` does not return an actual reference to the created actor.) The test driver then asynchronously sends a message to the Client actor using ActorFoundry's `send` method to initiate the exploration. The `send` method takes as arguments an `ActorName` indicating the message's destination, a `String` indicating the message type (i.e., the name of the method that will process the message in the receiving actor), and any arguments required for the specified message type. Since the message `start` is sent asynchronously to the client, the `test` method continues execution (in this case the method itself terminates right away, but the entire program keeps working as there are alive actors).

Our example `Server` actor simply stores and retrieves an integer value. (The actual server from the ScalaWiki code was keeping track of an inventory of items with specified prices and quantities.) In ActorFoundry, each actor is a subclass of the `Actor` class. Each actor can process a set of messages as denoted with the `@message` annotation on the appropriate methods. In addition to storing and retrieving the value, the `Server` actor can also process a `shutdown` message, which instructs this actor to terminate itself as it invokes the `destroy` method from ActorFoundry.

Our example `Client` actor communicates with the server. The client first sends to the server an asynchronous message to store the value 1. The client then sends a message to the server to

```
class Driver extends Actor {
    @message public void test(String[] args) {
        ActorName server = create(Server.class);
        ActorName client = create(Client.class, server);
        send(client, "start");
    }
}
class Server extends Actor {
    int value = 0;
    @message void set(int v) {
        value = v;
    }
    @message int get() {
        return value;
    }
    @message void shutdown() {
        destroy("server has finished processing");
    }
}
class Client extends Actor {
    ActorName server;
    Client(ActorName s) {
        server = s;
    }
    @message void start() {
        send(server, "set", 5);
        int v1 = call(server, "get");
        System.out.println("value v1 = " + v1);
        int v2 = call(server, "get");
        System.out.println("value v2 = " + v1);
        // assert (v1 == v2);
        send(server, "shutdown");
    }
}
```

Figure 3.1: Example code using ActorFoundry

retrieve the value, using the `call` method for *synchronous* remote procedure call. Namely, after the client sends the `get` message, it blocks until it receives a reply from the server and then stores the return value into the appropriate variable. Note that the client retrieves the value twice, and in our example code, compares the return values when the `assert` is uncommented. (The actual code from ScalaWiki was also retrieving the inventory of items twice but performing two different computations on the inventory.) The client finally terminates the server.

The cause of problems in this example is the *order of message deliveries*. In general, actor systems do not guarantee in-order delivery of the messages; in other words, the default commu-

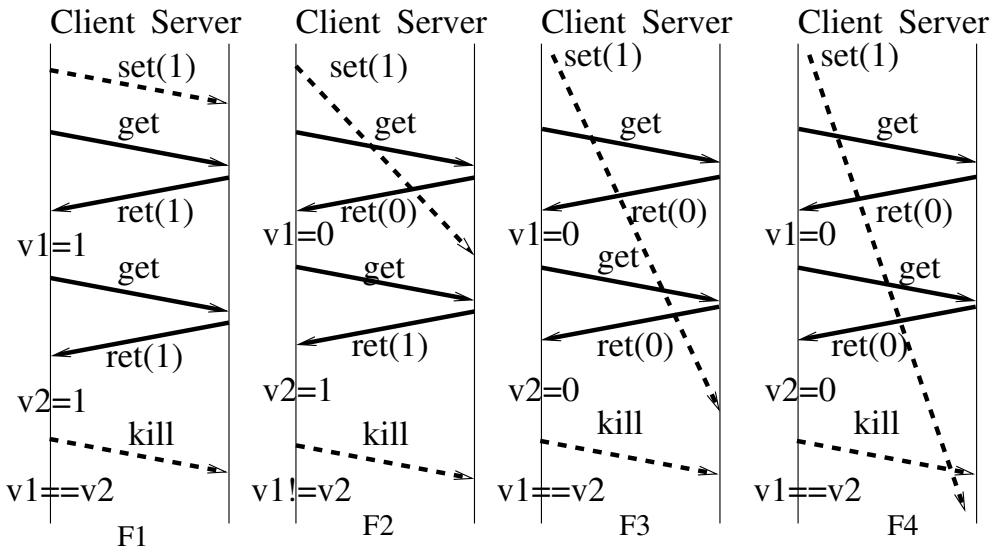

Figure 3.2: Possible message schedules and final states: correct, incorrect, correct, warning

nication channels between the actors are *not FIFO*. However, anecdotal experience shows that assuming in-order delivery is a common cause of programming errors in many actor programs. In our example, for instance, the server need not process the message `set` before the first message `get`. Figure 3.2 shows some possible executions for our example program. Specifically, if the server interleaves processing of `set` between the two `get` messages, it will return inconsistent values for `v1` and `v2`. While this execution is not very likely (e.g., we ran our example code on the standard Scala run-time 1000 times, and it always processed `set` before the first `get`), it is possible. The program therefore has an *atomicity violation*. (The Scala developers confirmed the bug that we reported for their code from ScalaWiki.)

Given an actor program, Basset can explore (all) different program executions due to different message orderings. Basset starts the execution from the `test` driver and explores non-deterministic choices that arise when several messages can be delivered. Basset also provides two well known optimizations that can prune exploration: dynamic partial-order reduction [SA06, CGP99] (see Section 3.5.2) and state comparison [SL08,CGP99] (see Section 3.5.1). For this example, we discuss how Basset performs a stateful search using Basset's customized state comparison for pruning.

Figure 3.3 shows the state space that Basset explores for our example code. Each state of an actor program consists of the *state of actors* (in our example, the field `value` in the server and the variables `v1` and `v2` in the client) and a *message cloud* (with all messages that have been sent

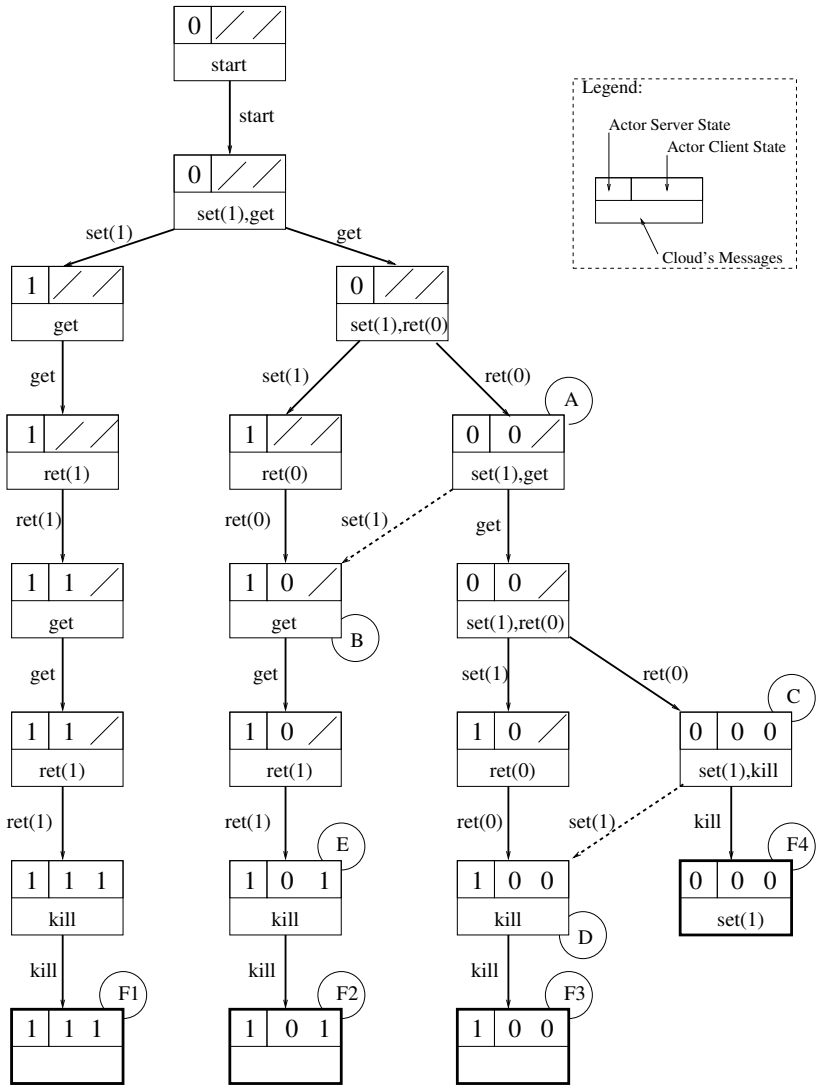

Figure 3.3: State space for the example program

but not yet delivered and can be thus delivered in any order). In Figure 3.3, a slash denotes an undefined state variable. Note also the `ret` messages that represent the return values of synchronous calls; these messages are not explicitly visible in the code, but the ActorFoundry library internally handles sending them.

The state space has 21 different states (including the starting state). We have labeled some of these states (e.g., A, B... E, F1, F2...) to be able to refer to them in the following text. The edges between states represent transitions that process messages. Each edge is labeled with the message that is delivered for that transition. Each box contains information about the state of the `Server` and `Client` actors as well as a list of the messages that have been sent, but not yet delivered. As

shown in Figure 3.3, the `Server` actor's state consists of the `value` field and the `Client` actor state consists of the `v1` and `v2` variables.

In this example, Basset explores 21 states and finds *two potential bugs*. In the state labeled `E` (for "error") in Figure 3.3, the values of `v1` and `v2` differ, being 0 and 1, respectively. (If the `assert` is uncommented, Basset would report a bug and stop the exploration for that path.) Additionally, in the state labeled `F4` (for "final state 4"), Basset generates a *warning* since the server actor is not alive while there are undelivered messages for that actor. This case occurs when the server processes `shutdown` before `set`, as shown in Figure 3.2.

Note that Basset explores only 21 states for this example due to a stateful search that compares states using a custom comparison for actor programs. This comparison allows Basset to detect that states `B` and `D` can each be visited through more than one execution. Without the comparison, Basset would explore both the subpath from `B` to `F2` and the subpath from `D` to `F3` twice during the exploration, resulting in a total of 27 explored states.

As an alternative to stateful exploration, Basset also supports dynamic partial-order reduction for actor programs. For this example, using the dCUTE partial-order reduction results in a total of 19 explored states when messages are considered for delivery to actors in the order in which the recipient actors were created. It is interesting to note that that if messages are considered for delivery in the reverse order, there would be no reduction in the number of transitions explored. This dependency on selection order is discussed further in Chapter 4.

We discuss Basset's state comparison and dynamic partial-order reduction capabilities in more detail in Section 3.5 and our experiments with these techniques in Section 3.10.

3.2 Basset Architecture

In Section 1.3 we briefly introduced the Basset testing framework and the two tool instantiations that were developed for ActorFoundry and Scala. This section provides an overview of Basset's architecture including the common, language-independent functionality for running actor programs and exploring different message schedules.

As we developed Basset, there were several objectives we tried to keep in mind. Naturally, these objectives affected the design decisions we made for the various aspects of the software.

Figure 3.4: Basset systematic testing environment software layers

A discussion of some of these objectives follows: For example, one objective was to minimize changes to the application code to be tested. In our experience, we have been largely successful in this endeavor. Actor application code can be run without changes using Basset. It is, however, necessary to write short test drivers, similar to those a developer might create using a testing framework like JUnit (`http://junit.org`). Of course, test drivers like these would be necessary for any approach for testing actors. Such testing would require creating a scenario with one or more actors and sending them messages to process.

A second objective was to provide a platform that would support *efficient testing and state-space exploration*. There are two elements to this objective: First, we were willing to forgo fast straightline execution of actor code, in favor of more efficient state-space exploration. Second, we chose to focus our efforts on the testing of user-written application code rather than consider both the applications and the underlying runtime architectures on which they execute. Our initial experiments showed that complete exploration of an actor application including its runtime library was impractical. The runtime libraries underlying the user-written code are complex multithreaded libraries that effectively prohibited the use of tools such as Java PathFinder (JPF). So as to allow

more complete testing of application code written by the developers, we chose to remove the runtime architectures from consideration. We realize, of course, that by doing so our testing may not expose all errors that could occur in a production environment. However, we are able to discover problems that are due to errors in the application code. This approach is analogous in some ways to how a tool like JPF tests user applications written in Java. JPF is itself a Java Virtual Machine (JVM). By performing testing using JPF, testers are not running their applications on the actual JVM that would be used in production, and may miss errors inherent in the JVM. Similarly, the Basset core software is an abstraction of the complex actor runtime architectures that will be used in production environments.

A third objective was to facilitate the reuse of testing techniques and capabilities across multiple languages. Specifically, we wanted capabilities such as dynamic partial-order reduction to be available to all instantiations of the framework. Again, our efforts here were largely successful. This does not mean that all such capabilities are always entirely transparent to the different instantiations, but efforts were made to do so where possible, and to keep any code necessary in the adaptation layers to a minimum.

As discussed earlier, we view the Basset testing environment as having four primary layers. Figure 3.4 displays a high level view of these layers. The base layer of our software environment is the *Java PathFinder* (JPF) model checker (described in Section 2.5). The *Basset Core* provides a common set of systematic testing capabilities that are built as an extension to JPF and handle the special requirements of systematically exploring the behavior of actor programs. *Actor programs* developed using ActorFoundry or Scala interface with the Basset core through the use of a *language adapter* which redirects actor-related operations to the Basset core instead of the language's normal runtime library. The sections that follow present these layers and other aspects of the Basset framework and its instantiations in more detail.

3.3 Actor Program Layer

At the top of our software stack is the *actor program* to be tested. As discussed above, one of objectives when designing our framework was to be able to systematically test actor programs without changes to the source code. Basset allows actor programs to be written normally in the

supported languages (i.e., ActorFoundry or Scala). It is *not* necessary to alter or instrument the code to explore its behavior with our framework.

We have been successful in our efforts to do this to the degree that the testing of a given subject does not attempt to ascertain the internal state of the actors. Note that Basset's limitations in this respect are no different than would be encountered with other testing approaches. For example, ActorFoundry does not expose actual actor references to other actors; only the runtime architecture can access the actor object directly. ActorFoundry facilitates actor mobility by using `ActorName` objects to provide a level of indirection. Since Basset effectively replaces ActorFoundry's runtime library during testing, it should not be overly difficult to expose the actual `Actor` objects to our test drivers, thus allowing the use of Java reflection to inspect an actor's internal state. We leave this as future work for Basset.

3.3.1 Drivers

Although users do not need to change the source code of their programs, they do need to write test drivers to facilitate Basset's exploration of the subject. Note, however, that similar driver code would be necessary for any unit testing of actor applications, so this is not entirely specific to Basset. The test drivers can be as simple as the one shown in Figure 3.1. This particular driver is written for a simple ActorFoundry program and is itself an ActorFoundry actor. It can receive just a single message type, `test`, that creates two test subject actors and then sends a single message to the client actor to start execution of the program.

We also support the creation of slightly more complex drivers. In the simple example above, all of the code in the `test` message is subject to exploration by Basset. This is fine for this simple example since the creation of the actors themselves does not involve sending messages. However, actors are often used to represent open systems (i.e., systems that interact with their environment). For these systems, it may be the case that we wish to explore the behavior of the program as it reacts to "external" messages *after* the system has been started and reached a steady-state. Since the messages sent from inside the `test` method (including messages sent by actor constructors) will be systematically explored in their entirety, any start-up processing related messages would also be explored. Exploring these messages combined with "external" test messages could easily result

in a huge state space whose exploration would be both impractical and undesirable.

To remedy this problem, drivers can be created with an additional `setUp` method. Basset does *not* explore the interleaving of messages sent as a result of executing the code in the `setUp` method. Actors are created and messages are processed in order of creation until there are no deliverable messages remaining (i.e., the program has reached a steady state). Only then does Basset send a `test` message to the driver to begin exploration.

3.4 Basset Core

The Basset core is the heart of our systematic testing framework. Effectively, it is an abstraction of the complex runtime architectures included with many actor systems. However, rather than supporting just a single execution of an actor program, the core supports the systematic exploration of message delivery schedules.

When testing programs with Basset, language adapters developed for the actor systems supported by Basset (currently ActorFoundry and Scala) are used to redirect actor related API calls to the Basset core. Basset manages the overall exploration of possible message delivery schedules by handling the following functions: *actor state management* (e.g., keeping track of created and destroyed actors), *actor execution* (e.g., managing actor objects and controlling thread switching), and *message scheduling and management* (e.g., handling the details of scheduling and delivering messages, such as tracking message causality when partial-order reductions are used). In this section, we first provide more details about these three functions, then discuss our overall exploration algorithm, and finally present Basset's error detection facilities.

3.4.1 Actor State Management

Each actor program creates several actors that compute and exchange messages. Various actor libraries provide different specialized behavior for actors. The Basset core therefore does not create the concrete actors by itself but delegates that task to particular instantiations. The core only maintains generic information about actors, keeping track of all actors created and destroyed during an execution, and the status for each actor, which can be one of the following: executing, waiting for a general message, or blocked waiting on a reply for synchronous message (such as the

`call` method shown in Figure 3.1). Basset uses all this information for efficient message scheduling (Section 3.4.3), during state comparison to prune redundant execution segments (Section 3.5.1), and to facilitate deadlock detection (Section 3.4.5).

3.4.2 Actor Execution

A critical aspect of any actor system is how to execute the actor code that processes each message. Semantically, each actor has its own thread of control. However, efficient implementations of actor libraries [HO07, SM08] typically do not assign one native thread/process per actor and do not create a new thread whenever a new actor is created, since these operations are expensive; instead, they employ thread pools to reuse threads for new actors, migrate actors among threads, and/or use more lightweight parallel constructs, such as the Java Fork/Join Framework [Lea00] from the `java.util.concurrent` package in the Java 7 version of the standard Java library. ActorFoundry, for instance, uses a continuation-based thread pool approach based on Kilim [SM08]. And Scala also utilizes a thread pool as part of its approach to unifying threads and events [HO07].

Since Basset aims for efficient *exploration* (not *execution*) of actor programs, Basset uses a separate `ActorThread` object/thread for each actor. Exploring all possible fine-grain interleavings of instructions from these threads would be very costly (see Section 3.10.1) and is not necessary when actors have no shared state. Some actor libraries allow state sharing but most avoid it, say, by using functional languages (e.g., Erlang [Arm07]), pass-by-copy rather than pass-by-reference messages (e.g., ActorFoundry [AF]), or a type system based on linear types to statically check for absence of sharing. Hence, Basset uses the *macro-step semantic* [AMST97, SA06] for actor execution: after Basset delivers a message to an actor, the actor executes atomically until the next receive point (which either encounters a synchronous call or finishes the processing of the message and waits for a new message). The soundness of macro-step semantics is discussed elsewhere [AMST97]. Similarly as for creating actors, the Basset core does not itself create actor threads but delegates that task to particular instantiations, which provide the main control for the processing of one message.

3.4.3 Message Management and Scheduling

Central to Basset's exploration capabilities are its message management and scheduling functions. Actors communicate by exchanging messages. Basset again delegates the creation of concrete messages to instantiations, but it maintains a *message cloud* of all messages that were sent but not yet delivered to actors. The main loop in Basset controls the delivery schedule of these messages. Whenever the cloud contains more than one deliverable message, Basset non-deterministically chooses to deliver one of these messages to its receiver actor. Basset then systematically explores all possible program executions that arise from the delivery of the messages in the cloud, using either state comparison (Section 3.5.1) or dynamic partial-order reduction (Section 3.5.2) to prune the exploration. Currently the use of state comparison and dynamic partial-order reduction in Basset are independent and mutually exclusive. However, recent work [YCGK08, YWY06] proposes combining the two techniques, and we leave it as future work to explore this possibility for Basset.

It is important to point out that not all messages are deliverable at all times. One reason is that an actor can terminate itself (e.g., using `destroy` in ActorFoundry as shown in Figure 3.1) while there are still sent but undelivered messages for that actor. In this case, the message can never be delivered, and Basset reports this as a warning of potential program error. Another reason is that most actor libraries allow the specification of *constraints* indicating the messages that actors can receive. Scala, for instance, expresses these constraints by pattern matching on the messages [OSV08]. If a message sent to an actor does not match any of its patterns, the message cannot be delivered until the behavior of the actor changes such that its new pattern matching accepts the message.

Basset delegates state management functions (e.g., state restoration for backtracking) to the underlying JPF tool on which Basset is built. Section 2.5 provides more detail regarding the capabilities provided by JPF.

3.4.4 Exploration Algorithm

Figure 3.5 shows the pseudo code of the algorithm that Basset uses to explore all possible message delivery schedules for an actor program (modulo pruning based on state matching or DPOR). The algorithm maintains two sets of program states: states that need to be explored and states that

```
StatesToExplore = { initial program state } // singleton
VisitedStates = {} // empty set of states

while (StatesToExplore is not empty) {
    State = choose a state from StatesToExplore
    StatesToExplore = StatesToExplore − {State}
    DeliverableMsgs = filter deliverable messages from State.Cloud
    for each (Msg in DeliverableMsgs)
        Cloud = State.Cloud // set of message objects
        Cloud = Cloud − {Msg}
        Actors = State.Actor // set of actor objects
        NewMsgs, NewActors = process Msg by the Msg.Receiver actor
        // processing can send new messages, create new actors,
        // and change the local state of the receiver actor
        // (including destroying the actor itself)
        Cloud = Cloud ∪ NewMsgs
        Actors = Actors ∪ NewActors
        NewState = pair of Cloud and Actors
        if (NewState ∉ VisitedStates) {
            VisitedStates = VisitedStates ∪ {NewState}
            StatesToExplore = StatesToExplore ∪ {NewState}
        }
    }
}
```

Figure 3.5: Pseudo-code for exploration in Basset

have already been visited. The set of `StatesToExplore` initially contains only the starting state. The algorithm first non-deterministically selects a `State` to explore and removes it from the set `StatesToExplore`. Every `State` is a pair of `Cloud` (of messages) and the state of `Actors`. From the `Cloud`, the algorithm selects all `DeliverableMsgs` that can be delivered to the `Actors` in the current state. (Recall that some messages cannot be delivered due to constraints or terminated actors.) Then, the algorithm iteratively selects a message from this set, removes it from the cloud, and delivers it to the receiver actor. At this point, Basset gives control to the receiver actor so that it can process the message. While processing the message, the actor can modify its internal state, send messages to actors (including itself), create new actors, or destroy itself (if the language allows this). Upon encountering a message receive (implicit, either in a synchronous `call` or in waiting to process a new message), the actor gives the control back to the exploration loop. Any modification on the program state performed by the executing actor (e.g., new messages sent, new

actors created, the actor's state changed, or the actor's self destruction) are reflected in the current `Cloud` and `Actors` that together form the `NewState`. If this `NewState` has not already been visited, it is added to both `VisitedStates` and `StatesToExplore`. The exploration then continues until there are no states to be explored.

3.4.5 Error Checking

Basset provides several general checks for executions of actor programs. As illustrated in Section 3.1, Basset can be used to check for *state assertions* (expressed using arbitrary Java expressions) and for *undeliverable messages* at the end of an execution path (due to actors being terminated or blocked). Basset can also detect *deadlocks*. An obvious deadlock occurs when several actors are blocked, each waiting for another (in a cycle) to return from a synchronous call. Another type of deadlock can occur when the execution reaches a final program state where no alive actor can make progress. Since actors are often used to develop open systems, such a final state is not necessarily a deadlock; it may be that actors are waiting for a new message from the environment [Agh86, AMST97, SA06]. To check for deadlocks in such cases, Basset allows the user to "close" the system by providing a model of the environment (as another actor).

The developer can decide which environment best models an error for each specific program, thus providing greater flexibility. Basset defines two built-in environments that can be used to check the user program:

- TOTAL: the environment can send any new message (except a return from a synchronous call). An error is reported only if there is some actor that is blocked because of a synchronous communication. Any other idle (non-terminated) actor will not be considered to be in error. This is the default behavior and the user code does not need to be modified to be checked under this condition.

- EMPTY: The environment can send no new messages. An error is reported if at the end of an execution there exists some alive (either blocked or idle) actor. In other words, this environment expects all actors to have terminated at the end of the program execution.

Beyond this, the developer is given the ability to provide a customized definition of the envi-

ronment, which can be used to finely model the states for which an actor has to be considered in an error.

3.5 Basset Core Optimizations

One of the key problems with systematic testing and explicit state model checking is the "state-space explosion" problem. The non-determinism inherent in concurrent systems such as actor programs often results in state spaces that can be extremely large and impractical to explore in a brute force manner. Two approaches to helping alleviate the problem are stateful search which uses state comparison to identify previously visited states and dynamic partial-order reduction (DPOR) which identifies redundant orderings of transitions. Both approaches seek to identify and prune exploration paths that do not need to be explored. In Sections 3.5.1 and 3.5.2 we briefly describe Basset's state comparison and DPOR capabilities. And in Section 3.5.3 we consider the effect of combining multiple transitions/steps into larger steps

3.5.1 State Comparison

Basset can perform a stateful exploration, checking whether a new state has been already visited previously, effectively comparing one state against a set of states. This is a standard operation in explicit-state model checking [CGP99]. A challenge for object-oriented programs (whose state include heaps with connected objects) is that states need to be compared for *isomorphism* [Ios01, MD05, BKM02]. Namely, two states are equivalent when their heaps have the same shape among connected objects and the same primitive values, even if they have different object identities. Typical comparison of states for isomorphism involves linearizing the *entire* states into an integer sequence that normalizes object identities such that isomorphic states have equal sequences [Ios01, MD05]. The JPF tool, on top of which Basset is implemented, provides such state comparison.

In addition to JPF's default state comparison, Basset provides a custom comparison that has been specialized for the actor domain. For example, an additional challenge for actors is that the top-level state items—actors and message clouds—are sets. The usual linearization does not specially treat sets but simply compares them at the concrete level at which they are implemented (say, as arrays or lists) and thus could find two sets with the same elements to be different because

of the order of their elements. To compare states of actor programs, we provided in Basset a known heuristic that first sorts set elements and then linearizes them as usual [d'A07]. This heuristic offers more opportunity to identify equivalent sets and states but does not guarantee all equivalent states will be found: the sorting is done only for the elements without following any pointers from these elements, and thus does not handle arbitrary graph isomorphism [SL08].

3.5.2 Partial-Order Reduction

As an alternative to performing stateful exploration with path pruning, we adapted three dynamic partial-order reduction (DPOR) techniques to work with the framework. The first is a DPOR for actor programs proposed in dCUTE [SA06]. The second is an adaptation of a DPOR based on dynamically computing persistent sets [FG05]. The third combines our persistent set implementation with sleep sets [God91, God96]

These reductions use the *happens-before* relation [Lam78] to avoid executing message schedules that can be identified as equivalent. They identify situations where only a subset of the messages available for delivery need be considered when non-deterministically choosing which message to deliver next. To facilitate the use of these partial-order reductions, we extend the actor and message representations to optionally include vector clocks, which can be used to efficiently track the happens-before relation for message send and receive events [Fid88, SA06]. Since the benefit of these partial-order reductions is sensitive to the order in which messages (and their receiving actors) are considered for delivery, Basset provides eight different orderings. Our experiments using DPOR are briefly discussed in Section 3.10 and are presented in greater detail in Chapter 4.

3.5.3 Step Granularity

The primary source of non-determinism for actor programs is the order in which messages are delivered to the actors. When exploring different message schedules, the standard step used by Basset consists of all instructions starting from some actor receiving a message up to the next point where the same actor can receive a message. This step is called a *macro-step* [AMST97, SA06]. Provided that an actor program is limited to message passing only, no interleaving of different actor threads is necessary, i.e., all the behaviors of the program with fine-grained thread interleaving

can be obtained with the macro-step interleaving. Basset always explores macro-steps of actor programs.

An additional consideration is whether to merge macro-steps from several actors when there is only one message in the cloud. We refer to such steps that *combine several deterministic macro-steps* from different actors as *Big* steps. Recall the state space from Figure 3.3. After the `set(1)` message is executed from the starting state, the rest of the execution until the `F1` state is deterministic: there are several macro-steps alternatively executed by the server and client actors, but there is always one message in the cloud. A Big step would thus execute the program until `F1`, without performing state comparison for the intermediate states shown in the figure. In contrast, a *Little* step executes *only one macro-step at a time* and performs state comparison for each intermediate state. In other words, Figure 3.3 shows the exploration that Basset performs for Little step.

Whether Big or Little steps provide faster exploration is not immediately clear. The trade-off is that Big steps are faster for straightline execution since they perform longer executions without stopping to compare states, but Big steps can miss the opportunity to find equal states and thus end up re-executing a number of states, resulting in a slower overall exploration. For example, in the state space from Figure 3.3, Big steps would miss that states `B` and `D` are equal along two different execution paths and would thus end up re-executing twice the code from `B` to `F2` and from `D` to `F3`. On the other hand, Little steps have slower straightline execution and more frequent state comparison, which can find more opportunities to avoid re-execution but also runs the risk that frequent comparisons end up finding different states and only unnecessary slow down the exploration. Whether re-execution or state comparison is more costly depends on the particular execution platform and frequency with which states are repeated during state exploration. Our experiments with Java PathFinder and subject actor programs described in Section 3.10 show that Little step performs better than Big step.

3.6 JPF Implementation

We implemented our Basset framework as an extension to Java PathFinder (JPF), an extensible, explicit-state model checker for Java [VHB+03, JPF]. We provided a general overview of JPF's capabilities in Section 2.5. In this section we provide specifics on how we developed Basset to

interact with JPF, along with a description of changes that we made to the JPF core.

The key extension we made to JPF is in the control of thread scheduling. Recall from Section 3.4.2 that the Basset architecture puts each actor in its own thread. The actor code itself is in Java (more precisely, compiled to Java bytecode). Additionally, Basset has a main controller thread that decides which actor(s) should be executed at which point. We wrote the main controller itself in Java so that it also runs on JPF's JVM (and not on the host JVM). All these threads are actual JPF/Java threads. Based on the macro-step semantics (Section 3.4.2) it is not necessary to explore all fine-grained interleavings of these threads; the non-determinism in actor programs is due to the order in which messages are processed by the actors. Therefore, Basset enables only one of these threads at a time.

Note that the thread switch could *not* be efficiently implemented in pure Java executing on JPF's JVM. To ensure that execution properly switched back and forth between actor threads and Basset's main controller thread, we needed greater control over thread switches than the Java language supports. Namely, the main loop in Basset proceeds as follows: Basset's main controller thread chooses one actor to execute (more specifically, one message to deliver to an actor that then starts processing the message), and when that actor *blocks* (waiting to receive a message), the main controller should execute to schedule another actor. However, once the actor blocks, it cannot explicitly return control to the main controller at the Java level.

For exploration purposes, we do not want more than one thread enabled at any given time. We want to seamlessly switch back and forth between Basset's main controller thread and whichever actor thread needs to process a message. Effectively, we wanted to *efficiently* and *atomically* perform three actions: disable the current thread, enable a disabled thread, and yield control to the newly enabled thread. We also wanted to combine this set of actions into a single scheduling event from the perspective of JPF.

To accomplish this, we extended JPF, using JPF's MJI interface to implement our own atomic thread switches as described above. In effect, this simplified the task performed by JPF's default thread scheduler, as there would never be more than one thread enabled at any given time.

The only change we made which directly impacted JPF's core code was to eliminate the creation of JPF backtracking points when switching back and forth between the actor threads and the main

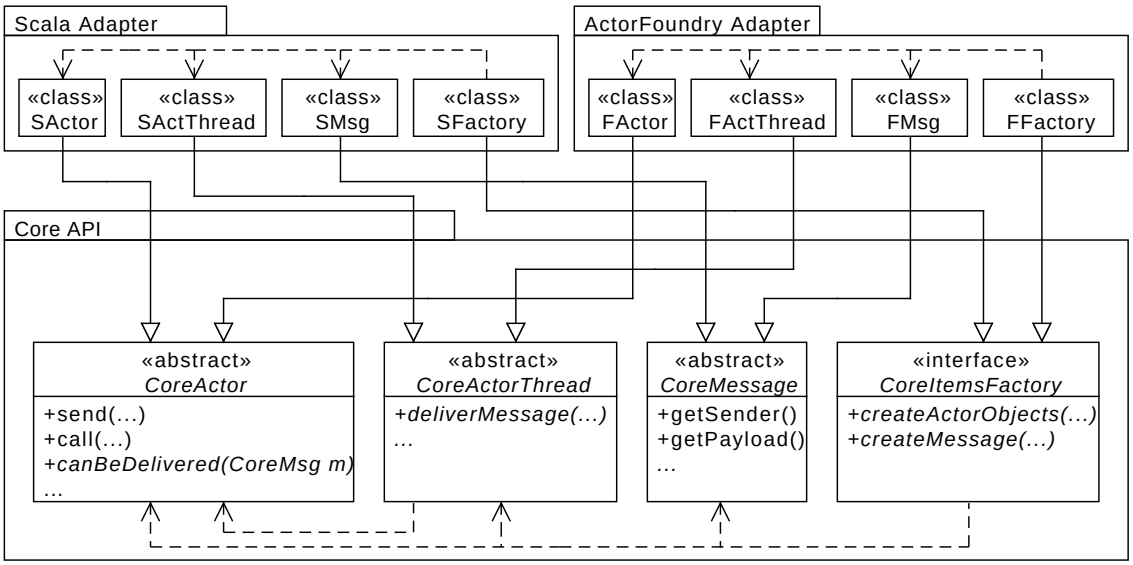

Figure 3.6: UML class diagram for language adaptation layers

controller thread. Since the JPF main loop is structured around executing only one thread in a transition, we modified the JPF core code to enable longer transitions. This allows us to consider the selection and delivery of a message by Basset's controller thread and the processing of that message by an application's actor thread as a single transition. To the best of our knowledge, this is the first JPF extension that considered state transitions with bytecodes executed by more than one Java thread.

3.7 Framework Instantiations

We next describe our instantiations of Basset for the ActorFoundry and Scala libraries. *By instantiating the framework for a library, we effectively create a new state-space exploration tool for that library.* That tool automatically has all of the capabilities that are built into the framework, such as state comparison or partial-order reduction.

In each case, we started from the existing actor library and modified/simplified it with the following goals: (1) preserve the API of the library from the perspective of the actor programs (such that we can check unmodified actor applications); (2) simplify the library, considering that we want fast exploration (for relatively small program states) and not necessarily fast execution

(for relatively large program states, e.g., scaling up to thousands of actors); and (3) connect the library into the Basset framework to enable exploration. Our modifications removed some parts of libraries (e.g., distribution of actors across various computers, because our goal is to *check the actor applications, not libraries*, and distribution is an implementation feature of the library and not a semantic part of the applications) and replaced or modified other parts (e.g., the `Actor` class in ActorFoundry, as described below). While these modifications of libraries may appear time consuming from their description, they were actually much easier to perform than building the general Basset framework.

Figure 3.6 shows a UML class diagram for building part of software connecting the two instantiations into the Basset framework. The key entities that Basset manipulates are actors, actor threads, and messages. Basset does not directly create the objects for all these related entities but instead uses the Abstract Factory design pattern [GHJV95] or, in some cases, uses modified versions of the target language's library code. During execution, the Basset core itself refers to the `IItemsFactory` interface, and each instantiation provides concrete classes that can be used to create appropriate entities. The concrete instantiation classes implement the abstract methods from the core classes, i.e., `canBeDelivered` and `deliverMessage`. In the next section, we provide a more detailed look at the implementation of our ActorFoundry instantiation as a means of shedding light on how one might build a language adaptation layer for a different actor language.

The changes required to create this adaptation layer are not large, but they required a deep understanding of both the ActorFoundry implementation and how Basset worked internally. However, the code provided here (along with the code for the Scala language adapter) should serve as a good model for the development of future framework instantiations. The code for both the ActorFoundry and Scala instantiations is publicly available as part of the Basset software. It can be downloaded from either the NASA JPF website [JPF] or the Basset homepage [Bas]. In Section 3.9 we comment on some of the differences encountered while developing the Scala instantiation. We do not, however, go into the same level of detail as we do for ActorFoundry.

3.8 ActorFoundry

ActorFoundry is a Java library, and it was fairly straightforward to connect it to Basset. To create the adaptation layer for this language, we performed the following tasks: (1) we modified several classes in the ActorFoundry library, (2) we created a `FoundryItemsFactory` class for use by Basset's core, and (3) we modified the startup class used to appropriately execute and test ActorFoundry programs in Basset.

3.8.1 Library Class Modifications

The Basset core manipulates several actor program entities during testing. Specifically, it keeps track of actors, actor threads, and messages. Basset does not need to know all of the specifics of these entities for particular instantiations. Rather, it manipulates them solely through the Java interfaces that expose the information it is concerned about. The interfaces are `IActor`, `IActorName`, `IActorThread`, and `IMessage`. Each of these interfaces is implemented by a corresponding Java class (i.e., `CoreActor`, `CoreActorName`, `CoreActorThread`, and `CoreMessage`) that contains fields and methods used by Basset to manage the exploration of an actor program. For instance, both `CoreActorThread` and `CoreMessage` contain code related to the vector clocks used by Basset's dynamic partial-order reduction algorithms.

`CoreActor`, `CoreActorThread` and `CoreMessage` are abstract classes that must be implemented by a specific language implementation layer. `CoreActorName`, on the other hand, is a concrete class. These actor names are used to provide a level of indirection in some actor languages, which allows capabilities such as actor location transparency. This type of functionality is present in ActorFoundry but it is not used in our Scala instantiation. `CoreActorName` objects are also used internally within Basset.

To modify the ActorFoundry library for use with Basset, we modified six classes currently provided by the library and added one new class. As a result of these changes, many of the other classes that make up ActorFoundry were effectively "disconnected". In other words, our code changes either redirected interaction with certain components to the Basset core or, in a few small instances, disabled the functionality. Many other ActorFoundry classes continue to be used without any changes. Thus, to use the ActorFoundry instantiation, it is necessary to include

the ActorFoundry library in the Basset environment and to add our modified classes before the ActorFoundry library.

The ActorFoundry capabilities that we disabled were primarily related to the runtime library's support for distributed actor systems. For example, ActorFoundry supports the migration of actors from one machine to another. This capability is largely transparent from the perspective of the application code that developers write. Since Basset's focus is the testing of that application code, we did not need to support these capabilities.

We next discuss the specific modifications and additions made to the ActorFoundry library. The six library classes that were modified or replaced are: `osl.manager.basic.BasicActorImpl`, `osl.manager.Actor`, `osl.manager.ActorImpl`, `osl.manager.ActorMsgRequest`, `osl.manager.ActorName` and `osl.util.DeepCopy`. In addition to these modified classes, we also added one new class: `osl.manager.ActorMessage`.

Actor Class

ActorFoundry does expose actors in the API (more precisely in the internal API used within the library), so we had to preserve that the actor class be named `Actor`. Since this class had no superclasses in the original library, we could easily turn it into a subclass of the Basset actor class (i.e., `CoreActor`) and then modify it to provide both functionality required by the ActorFoundry library as well as that required by the Basset framework. There was no specific constructor for Actor, so we added one with a single statement: `super(new ActorName());` where `ActorName` is a class from ActorFoundry.

The `CoreActor` class includes methods to query and update the status of an actor that Basset maintains. At any point during the execution of an actor program, a given actor has one of the following Basset defined states: EMPTY (the state of an actor before it is fully created), SUSPENDED (when an actor's constructor is executed), ACTIVE (when an actor is processing a message), WAITING (when an actor is waiting for a message to be received), WAITING_ON_REPLY (when an actor is waiting for a reply message for a synchronized RPC-style call), TERMINATED (when an actor terminates its computation – in Scala, such actors can be restarted) and DESTROYED (when an actor has been deleted).

```
class Actor extends CoreActor {
   ...
   @Override public boolean canBeDelivered(IMessage imsg) {
       ActorMessage msg = (ActorMessage) imsg;
       if (isDestroyed())
           return false;
       if (isWaitingOnReply() && msg.isReturnMessage())
           return true;
       return isWaiting() && !isDisabled(imsg);
   }

   // checks if the actor's method to be called is disabled
   private boolean isDisabled(IMessage imsg) {
       ...
   }
   ...
}
```

Figure 3.7: ActorFoundry's `canBeDelivered` method

An important part of overriding the `CoreActor` class is creating a concrete version of the `canBeDelivered` method. The purpose of this method is to tell Basset whether or not a specific message can be delivered to the actor. For ActorFoundry, the answer is based on both the state of the actor (which is tracked by Basset) and whether or not ActorFoundry has enabled any constraints that would prevent delivery of the message. Normally, this check would be performed by ActorFoundry's scheduler. Since Basset is now performing the scheduler's functions, we need to provide the code to determine the deliverability of the message. To achieve this, we implemented the `canBeDelivered` method as shown if Figure 3.7. `canBeDelivered` calls methods provided in `CoreActor` to determine the status of the actor (see above) as well as an `isDisabled` method that was essentially copied from elsewhere in the ActorFoundry library.

As discussed above, we did not implement support for functionality such as ActorFoundry's distributed actors or node services. Although we did not change any code, the following methods in the `Actor` class are not supported by Basset: `migrate`, `cancelMigrate`, `extension`, `extensionImpl`, `invokeService`, `invokeServiceImpl`.

ActorName Class

The `ActorName` class in ActorFoundry consists of a single field of type `Name` and methods for serialization and deserialization. `Name` objects are generated by ActorFoundry's name service and

include information required to manage distributed actors, such as physical address, etc. Since we were not interested in supporting these capabilities, we modified `ActorName` to just extend our `CoreActorName` class. `CoreActorName` is a simple class consisting of an `id` field, which is incremented each time a new actor is created, and a `String` (i.e., "JPF_Actor-" + id) used by Basset to display actor information during exploration. There is also an optional `String` field that can be used to display additional information. This field is not used for ActorFoundry, but it is used for the Scala language adapter.

ActorImpl and BasicActorImpl Classes

The `ActorImpl` class and its subclass `BasicActorImpl` constitute the second level of ActorFoundry's actor runtime implementation. The `Actor` class described above provides the interface that the programmer uses to create applications. The `BasicActorImpl` and `ActorImpl` classes, on the other hand, provide the interface to the `ActorManager`, translating user service requests (e.g., message requests) into actual service invocations on the `ActorManager`. Additionally, `ActorImpl` extends the `Task` class which is the lightweight thread implementation for Kilim [SM08]. ActorFoundry uses the Kilim library to provide an efficient thread pool-based approach for actor execution.

For exploration purposes, Basset associates each actor with its own thread rather than using a thread pool approach. The switching of threads in the Java PathFinder JVM is not particularly expensive as it would be in a regular multithreaded JVM. To accomplish this, we changed `ActorImpl` to extend `CoreActorThread` (which itself extends the Java `Thread` class) instead of `Task`. The only other changes to `ActorImpl` were quite minor. We changed the protection level on the `actorInitialize` method from protected to public so that it could be accessed directly from Basset, and we added a simple constructor for the class:

```
public ActorImpl(Actor actor) {super(actor);}
```

While the changes to `ActorImpl` were minimal, the changes required for the `BasicActorImpl` class were more extensive. To begin with, `BasicActorImpl` also needed a simple constructor:

```
public ActorImpl(Actor actor) {super(actor); createMethodTable();}
```

The `createMethodTable` method and its supporting methods create a table of message types available in the actor. This is essentially a performance enhancement that front-loads the identification of

valid message types into the actor creation process.

We next replaced the `processMessage` method with a new `processDeliveredMessage` method. `processMessage` processed messages from a mail queue that is populated by `ActorFoundry`'s runtime architecture. The new `processDeliveredMessage` method accepts and processes a single message from the `received` field in `CoreActor`. As part of this change, the `execute` method has also effectively been disabled. Basset explores message interleavings and is now responsible for message delivery. Since Basset never delivers more than one message at a time to an actor, the execution loop provided by `execute` is no longer necessary.

The `implCall` method required several changes to interface with Basset. First, it was modified to create `ActorMessage` objects based on `ActorMsgRequests`. These messages are then sent using the `call` method in `CoreActorThread`. A larger change was required to the `implCall` method's handling of replies to the call. When waiting for a reply, ActorFoundry's `implCall` method searches its mail queue looking for either a reply message or an error message, ignoring other messages delivered to the queue. Our modified version of the code no longer has a mail queue, and Basset only transfers control to the actor if it has a reply or error message. This difference required this portion of the method to be rewritten. `implSend` method was also modified to create `ActorMessage` objects based on `ActorMsgRequests`. These messages are then sent using the `send` method in `CoreActorThread`.

It was also necessary to modify the `implDestroy` method to update the actor's status to DESTROYED (note: actors are never removed entirely in Basset) and the `implCreate` method to route the request to create a new actor to the Basset core rather than to ActorFoundry's actor manager. Note that the Basset core does not directly create the new actor; rather, it passes on the information to the `FoundryItemsFactory` described below.

Finally, several methods were disabled as their functionality is either not supported or was moved elsewhere. For example, we do not currently support the ActorFoundry's special actors for `stdin`, `stdout`, and `stderr` that are set up in the `initializeActor` method. The other code in this method was moved to the `FoundryItemsFactory` class. The `findConstructor` method was also moved to the `FoundryItemsFactory` class. Since we do not support actor migration, the `actorPostMigrateRebuild` was no longer needed. Also, the functionality provided by the `actorDeliver` method is subsumed by the Basset core.

```
public class FoundryItemsFactory implements IItemsFactory {

   public Object[] createActorObjects(Object ref) {
      ActorCreateRequest request = (ActorCreateRequest) ref;
      Class<?> actorClass = request.behToCreate;
      Object[] conArgs = request.constructorArgs;

      ActorName newActorName = null;
      Actor newActor = null;
      BasicActorImpl newActorThread = null;

      try {
         newActor = createActorClass(actorClass, conArgs);
         newActorName = (ActorName) newActor.getActorName();
         newActorThread = new BasicActorImpl((Actor) newActor);
         ActorCreateRequest copyReq = (ActorCreateRequest) request;
         newActorThread.actorInitialize(null, newActorName, copyReq);
         newActor._init(newActorThread);
      } catch (Exception e) {
         throw new RuntimeException(e.getMessage());
      }
      return new Object[] { newActorName, newActor, newActorThread };
   }

   protected Actor createActorClass(Class<?> actorClass, Object[] conArgs) {
      // create the actor behavior object.
      Actor newActor = null;
      try {
         if (conArgs == null || conArgs.length == 0) {
            newActor = (Actor) actorClass.newInstance();
         } else {
            Constructor<?> constructor = findConstructor(actorClass, conArgs);
            newActor = (Actor) constructor.newInstance(conArgs);
         }
      } catch (Exception e) {
         e.printStackTrace();
         throw new RuntimeException(e.getMessage());
      }
      return newActor;
   }

   private Constructor<?> findConstructor(Class<?> classType, Object[] args)
      throws NoSuchMethodException { <code omitted> }

   public IMessage createMessage(Object... args) {
      return null;
   }
}
```

Figure 3.8: ActorFoundry's `FoundryItemsFactory` class

ActorMessage and ActorMsgRequest Classes

By default, ActorFoundry creates a copy of a message's content when performing a `send` or `call`. To do this, ActorFoundry's `DeepCopy` class uses Java Serialization to create deep copies of a message's arguments. This Java feature is not supported by JPF at this time. Rather than attempt to extend JPF to handle this functionality, we developed an alternative approach that creates a deep copy of an arbitrary object using JPF's Model Java Interface (MJI) that recursively traverses each of the object's fields to create a replica. The resulting `MJICopier` class (located in the `gov.nasa.jpf.actor.util` package) was coded with efficiency in mind and is likely to have applicability beyond just the Basset framework. The `DeepCopy` class provided by ActorFoundry was replaced in its entirety by a simple class that uses this new `MJICopier` utility.

In ActorFoundry, the `Actor` class does not directly create a message. Rather, it creates an `ActorMsgRequest` which is passed on to the `implSend` or `implCall` method in `BasicActorImpl`. These methods then send the request to an actor manager that in turn handles actual message creation and processing. We could have changed `Actor` to directly create `ActorMessage` objects. Instead, we chose to handle the conversion in the `implSend` and `implCall` methods as described above in our discussion of the `ActorImpl` and `BasicActorImpl` classes. Although this decision to delay creation of the `ActorMessage` may seem arbitrary, it was actually consistent with our desire to minimize changes to the ActorFoundry code. Minimizing code changes was a guiding principle intended to make maintenance of the ActorFoundry adaptation layer easier if and when the ActorFoundry code base changes. The only changes we made to `ActorMsgRequest` were related to the new `DeepCopy` class described above. The `ActorMessage` class is an entirely new class that extends `CoreMessage`. It keeps track of all the information stored in `ActorMsgRequest` and provides support to allow Basset to more easily create return messages (for RPC-style calls) and error messages (i.e., a special form of return message used in error situations).

3.8.2 FoundryItemsFactory Class

The current interface for items factories specifies only two methods. The `createActorObjects` method is intended to return an array of three objects: one each of `CoreActorName`, `CoreActor`, and `CoreActorThread`. Originally, creation and initialization of these three objects were handled

```
public class Basset {
  ...
  if (language.equals(ACTOR_FOUNDRY)) {
      platform = new Platform(new FoundryItemsFactory());
      platform.setTestDriverName(subjectDriver);
      try {
          // create test driver actor
          String driverClassName = args[0];
          Class<?> driverClass = Class.forName(driverClassName);
          ActorName driver = PlatformUtil.createActor(driverClass);

          // create String array containing arguments for the driver
          String[] driverArgs = new String[args.length - 1];
          System.arraycopy(args, 1, driverArgs, 0, args.length - 1);

          // message interleavings are NOT explored during setup
          PlatformUtil.send(driver, "setUp", (Serializable) driverArgs);
          platform.setUp();

          // message interleavings are explored during test
          PlatformUtil.send(driver, "test", (Serializable) driverArgs);
          platform.test();
      } catch (Exception e) {
          e.printStackTrace();
      }
  }
  ...
}
```

Figure 3.9: Startup class modifications for ActorFoundry

by three separate factory methods. In practice, this turned out to be impractical due to the interdependencies among the three objects.

The `createMessage` method is intended to create `CoreActorMessage` objects. Typically, the factory methods are called by the `Platform` class, Basset's primary controller. However, we chose not to use the factory method for our ActorFoundry implementation (though we do for our Scala adapter). As discussed in section 3.8.1, we create messages in the `BasicActorImpl`'s `implSend` method. What is important is that we kept message creation in the adaptation layer and did not place language-specific code in the Basset core.

3.8.3 Startup Class

When executing Basset to perform tests, one must specify the language in which the subject program is written. This is done by specifying the appropriate runtime option. For example, to indicate

that the test subject is written using ActorFoundry, one would specify `+basset.language=foundry`. To run the simple server example in Figure 3.1 using Basset, one would type something like the following:

```
bin/jpf +basset.language=foundry gov.nasa.jpf.actor.Basset server.Driver
```

This would cause JPF to execute the main `Basset` class using the `server.Driver` class as a test driver. To support ActorFoundry we needed to update the `Basset` class to recognize the language value `foundry` and add code to launch ActorFoundry programs. The code we added to the Basset class creates an instance of the framework's `Platform` class and requests that it create a test driver actor (recall that the driver classes for test subjects are written as ActorFoundry actors). Code was also added to create and send a `setUp` message to the driver actor, followed by a `test` message. Figure 3.9 shows the important aspects of these changes.

3.9 Scala

Our support for Scala is based on the actor library from the standard Scala distribution. The changes we needed to make for Scala were somewhat different than those for ActorFoundry. This is primarily due to the Scala language's compilation model (from Scala source files to Java class files) and the rich API that the Scala actor library exposes to Scala applications. As with the ActorFoundry language adapter, the Scala adapter did not require the development of a large amount of new code. However, the development of that code did require an in-depth understanding of both the Scala and Basset codebases.

A key issue is that our adaptation layer for Scala needs to subclass the actor class from Basset and also needs to provide the actual interface of the Scala actor. To solve this issue, we used the Adapter design pattern [GHJV95] to translate the calls that a Scala application makes on an actor object into appropriate calls to the Basset actor. Moreover, it was not possible to add a field to the existing Scala `Actor` trait (a trait can be thought of as a Java abstract class) to point to its corresponding Basset actor, because of the way that Scala compiles trait's fields [OSV08]. Therefore, we maintain this correspondence as a map outside of the Scala `Actor` trait. Each actor operation translates calls from the Scala world into the Basset world using this map.

We do not go into details about the implementation of this instantiation in this dissertation.

However, as mentioned at the end of Section 3.7, the code for the Scala instantiation is available with the Basset distribution at either the NASA JPF website [JPF] or the Basset homepage [Bas].

3.10 Framework Experimental Results

We present several experiments using the Basset framework. We first briefly compare the state-space exploration of a trivial `helloworld` Scala application using Basset versus an exploration using the standard Scala library executing on JPF. We then describe the subject programs used to more quantitatively evaluate Basset for ActorFoundry and Scala. We finally present experimental results comparing the different state-space reduction options available in Basset. All experiments were performed using Sun's JVM 1.6.0_13-b03 on a 3.4GHz Pentium 4 workstation running Red Hat Enterprise Linux 5. We set the time limit to one hour, and we show partial results in cases where the exploration did not finish in an hour.

3.10.1 Basset Versus Original Library

As discussed in Chapter 1, one of the motivations for developing a framework specifically for actors was that general exploration tools can be very inefficient due to the complexity of how actor systems are implemented. To illustrate the increased efficiency obtained by exploring an actor program using Basset instead of directly exploring an actor program and its library running on JPF, we ran an experiment to compare performance of these two options. Recall that our goal is to check actor applications, not actor libraries. However, the libraries already exist in Java bytecode, so it is natural to ask whether we can run them on JPF. Specifically, instead of developing Basset, could we have taken a Scala application with the existing Scala library and run it directly on JPF? Our experiments show that such direct exploration is possible but *extremely slow*, even after several simplifications to the library. The reason is that the Scala library is a complex, multithreaded piece of code, and exploring it on JPF results in exploring a very large number of thread interleavings.

More concretely, we wrote in Scala a simple `helloworld` application shown in Figure 3.10. The program merely creates a single actor that prints `Hello World`. Running this code with the unmodified Scala library on JPF did not finish in an hour! We then simplified the library by: (1) removing a timer thread, (2) disabling actor garbage collection, and (3) reducing the size of

```
import scala.actors.Actor._

object HelloWorld {
    def main(args: Array[String]): Unit = {
        actor {
            System.out.println("Hello World")
        }
    }
}
```

Figure 3.10: Simple Scala helloworld program

the thread pool that the library uses to execute actors. JPF still took *over 7 minutes* to explore this application. In contrast, the Scala instantiation of Basset takes *a fraction of a second* for this application. The key reasons for this speedup are that Basset uses a simplified framework with the macro-step semantics [AMST97, SA06] for exploration and does not interleave executions from different threads, and it does not explore the complex code of the Scala library on which Scala applications typically run.

Though we did not perform a similar experiment for ActorFoundry, we would expect similar results in that JPF would not be able to effectively explore the original library. In fact, the library code for ActorFoundry contains network calls that the publicly available JPF does not even support as they depend on native code in standard Java libraries. Projects by Artho and Garoche [AG06] and Barlas and Bultan [BB07] provide solutions for modeling some of these calls, but the original ActorFoundry library would still have a prohibitively large number of thread interleavings. Using the simplified library in our Basset framework, thread interleavings are manageable for exploring interesting programs.

3.10.2 Subjects

Our Basset experiments use ten actor programs listed in Tables 3.1 and 3.2. The `server` subject is our running example described in Section 3.1. Three of the subjects implement more complex algorithms and were previously used in the dCUTE study [SA06]: `leader` is an implementation of a leader election algorithm; `spinsort` is a simple distributed sorting algorithm, and `shortpath` is an implementation of the Chandy-Misra's shortest path algorithm [CM82]. The `fibonacci` subject computes the n-th element in the Fibonacci sequence. `mergesort` and `quicksort` are implementa-

		ActorFoundry					Scala				
Experiment		Resources		Statistics			Resources		Statistics		
Subject	State Reduction	Time (sec)	Memory (MB)	# of States	# of Msgs Delivered	# of Execs	Time (sec)	Memory (MB)	# of States	# of Msgs Delivered	# of Execs
fibonacci	None	16	89	769	768	168	28	110	768	767	168
	JPF Comp.	9	65	188	299	9	11	82	80	144	4
	Actor Comp.	8	71	105	184	6	9	85	43	75	2
	DPOR-LCA	15	106	495	494	102	13	100	147	146	32
	DPOR-ECA	10	90	289	288	64	33	160	768	767	168
leader	None	24	119	1467	1466	374	44	122	1467	1466	374
	JPF Comp.	17	95	671	864	106	19	93	255	341	42
	Actor Comp.	15	94	465	651	73	16	94	187	237	34
	DPOR-LCA	21	147	911	910	188	30	165	667	666	138
	DPOR-ECA	14	113	493	492	101	28	119	612	611	126
mergesort	None	1888	474	113067	113066	33264	2316	419	113066	113065	33264
	JPF Comp.	559	359	21450	32770	3080	73	131	1054	2729	20
	Actor Comp.	197	205	6081	10909	727	25	117	270	719	4
	DPOR-LCA	6	72	40	39	8	9	82	39	38	8
	DPOR-ECA	11	86	212	211	54	17	105	211	210	54
philosophers	None	-	-	-	-	-	3601	426	215674	215673	64492
	JPF Comp.	-	-	-	-	-	492	248	12538	29943	468
	Actor Comp.	-	-	-	-	-	180	185	3504	8499	150
	DPOR-LCA	-	-	-	-	-	186	290	6168	6167	806
	DPOR-ECA	-	-	-	-	-	1528	378	64373	64372	9122
pi	None	2300	418	168646	168645	60480	3523	463	168645	168644	60480
	JPF Comp.	115	199	4436	7003	720	52	249	346	893	12
	Actor Comp.	60	156	1652	3376	120	39	240	188	554	4
	DPOR-LCA	22	130	734	733	105	37	197	733	732	105
	DPOR-ECA	10	86	166	165	24	15	136	165	164	24

Table 3.1: Comparing different state-space reduction techniques in Basset

		ActorFoundry					Scala				
Experiment		Resources		Statistics			Resources		Statistics		
Subject	State Reduction	Time (sec)	Memory (MB)	# of States	# of Msgs Delivered	# of Execs	Time (sec)	Memory (MB)	# of States	# of Msgs Delivered	# of Execs
quicksort	None	3601	522	176871	176870	38245	852	373	43438	43437	11088
	JPF Comp.	3601	516	83278	183734	3995	29	123	435	1196	5
	Actor Comp.	2075	465	42472	105877	2021	14	104	120	309	2
	DPOR-LCA	8	108	74	73	16	7	85	37	36	8
	DPOR-ECA	392	260	13281	13280	2772	33	164	912	911	210
scalawiki	None	-	-	-	-	-	640	372	22522	22521	4152
	JPF Comp.	-	-	-	-	-	215	207	6513	7303	1081
	Actor Comp.	-	-	-	-	-	179	217	5456	6267	836
	DPOR-LCA	-	-	-	-	-	923	382	22522	22521	4152
	DPOR-ECA	-	-	-	-	-	197	264	4644	4643	836
server	None	4	42	27	26	6	6	77	26	25	6
	JPF Comp.	6	41	24	24	5	6	75	23	23	5
	Actor Comp.	5	41	21	22	4	6	76	20	21	4
	DPOR-LCA	4	42	19	18	4	7	77	26	25	6
	DPOR-ECA	5	58	27	26	6	6	72	18	17	4
shortpath	None	178	238	10000	9999	3614	223	245	10000	9999	3614
	JPF Comp.	38	120	887	1772	140	35	126	534	1160	69
	Actor Comp.	18	110	261	608	28	20	115	230	556	22
	DPOR-LCA	13	89	287	286	98	16	114	287	286	98
	DPOR-ECA	50	154	1690	1689	408	58	212	1690	1689	408
spinsort	None	89	200	5046	5045	1152	118	234	5045	5044	1152
	JPF Comp.	22	99	528	861	31	18	103	317	508	24
	Actor Comp.	15	100	287	459	19	17	101	269	432	24
	DPOR-LCA	37	145	1290	1289	288	43	182	1289	1288	288
	DPOR-ECA	121	274	5046	5045	1152	145	181	5045	5044	1152

Table 3.2: Comparing different state-space reduction techniques in Basset (continued)

tions of distributed sorting algorithms that use a standard divide-and-conquer strategy to carry out the computation. `philosophers` is an implementation of the classic dining philosophers problem, where both the philosophers and the resources/forks are modeled as actors. `pi` is a porting of a publicly available [Pi] example for Message Passing Interface, which computes an approximation of the π number by splitting the task among a set of N worker actors. Finally, `scalawiki` is the original client-server application previously available from the ScalaWiki website. Our use of Basset exposed an atomicity violation in this code, which has been corrected in the latest version of the example. We did not translate the entire `scalawiki` from Scala into ActorFoundry but only translated the simplified `server`.

All of these subjects can be executed in the standard environments for Scala or ActorFoundry. No modification to the subjects' code was necessary to explore them using Basset. Several of these subjects (namely, `fibonacci`, `leader`, `mergesort`, `pi`, `pipesort`, `quicksort`, and `server`) are included as part of the publicly available release of Basset. The software can be downloaded from either the NASA JPF website [JPF] or the Basset homepage [Bas].

3.10.3 State-Space Reduction

In this section we illustrate the use of some of the framework's capabilities. Specifically, we present the results of experiments that were run using the different state-space reduction capabilities that are built into the framework. Our experiments use ten actor programs listed in Tables 3.1 and 3.2. In most cases, two versions of each subject were created: one using ActorFoundry and the other using the Scala language. The two exceptions are `scalawiki` and `philosophers`.

As discussed in Section 3.4, Basset provides two mechanisms for reducing the exploration of a state space: state comparison and dynamic partial-order reductions based on the happens-before relation. In addition to the default state comparison provided by JPF, we implemented a custom state comparison to improve the identification of previously visited states. The abstraction we use for state comparison allows for more aggressive pruning of redundant message schedules, which, in turn, results in faster state-space exploration.

Tables 3.1 and 3.2 show the results of experiments comparing JPF's standard state comparison (JPF Comp.), our custom actor state comparison (Actor Comp.), and the dCUTE dynamic partial-

order reduction adapted for actor programs and implemented in Basset (DPOR-LCA and DPOR-ECA). For reference purposes, results without state comparison or partial-order reduction (None) have also been provided.

For each type of state-space reduction, we tabulate the total exploration time in seconds, memory usage in MB, the number of states identified during the entire exploration, the total number of messages (across all executions) that were delivered during the exploration, and the total number of execution paths completed in their entirety (i.e., paths that reached a final state where no messages can be delivered). Effectively, the number of states and the number of messages are the number of nodes and the number of edges, respectively, in the state-space graph that Basset explores for a program. The variations in numbers between ActorFoundry and Scala are due to differences in how the subjects were implemented and how the drivers establish the initial state.

Execution time typically improves as we progress through the three types of state comparison, from None to the JPF comparison to the Actor comparison. In nearly all cases, the Actor comparison results in the fastest of these explorations. The single exception is for the ActorFoundry `server` experiments, where using no comparison was the fastest. In this case, we suspect the overhead of performing comparisons outweighed the potential savings for such a small number of executions. Memory utilization remains reasonable across all of the experiments, usually varying in line with the total number of explored states. Similar to reducing execution time, the Actor comparison reduces the number of explored states and the number of delivered messages. As the abstraction used by the state comparison is refined to consider only relevant state differences, the number of states and executions that can be pruned increases. As a result, the number of executions is not a particularly meaningful statistic when state comparison is used. The pruning of exploration can greatly reduce the number of executions that finish.

As previously mentioned in Section 1.4, the partial-order reduction is very sensitive to the order in which it chooses actors and messages for message delivery. To illustrate this, we ran experiments using two different orderings: DPOR-ECA delivers messages to actors based on the order in which the receiving actors were created (i.e., from earliest to most recent), while DPOR-LCA delivers messages in the reverse order (i.e., from most recent to earliest).

The results show that state comparison and partial-order reduction provide different speedups,

Experiment			Resources		Statistics		
Language	Subject	Step Size	Execution Time (sec)	Memory (MB)	# of States	Max Depth	# of Msgs Delivered
foundry	fib	Big	7	15	119	6	318
		Little	5	15	104	10	184
	leader	Big	19	16	433	7	995
		Little	13	16	464	11	651
	server	Big	2	10	10	4	26
		Little	2	10	20	6	22
	shortpath	Big	15	18	309	8	705
		Little	13	18	260	10	608
	spinsort	Big	13	18	247	7	599
		Little	10	18	286	10	459
scala	fib	Big	5	25	40	6	109
		Little	5	24	42	9	75
	leader	Big	10	28	131	7	308
		Little	8	26	186	11	237
	server	Big	3	23	10	4	25
		Little	3	23	19	5	21
	shortpath	Big	20	28	274	8	639
		Little	18	28	229	10	556
	spinsort	Big	16	27	220	7	552
		Little	13	27	268	9	432

Table 3.3: Comparing step granularity in Basset – *Big* vs. *Little* steps

with one or the other being better for different subjects. The differences in results between DPOR-LCA and DPOR-ECA illustrate that it is worthwhile to investigate heuristics for determining orderings of messages for actors. Our work along these lines is presented in Chapter 4. We believe that Basset provides an excellent research platform for such experiments on state-space exploration of actor programs, in the same way that JPF has provided an excellent research platform for state-space exploration of sequential and multithreaded Java programs.

3.10.4 Step Granularity

In Section 3.5.3, we discussed the trade-off associated with performing explorations using *Big* steps versus *Little* steps. We performed experiments to compare the performance of state-space explorations that enforced Little steps with explorations that allowed combining multiple Little steps into Big steps when appropriate. Each experiment was performed using our optimized Actor

state comparison (since the previous sections show it to be the fastest of all state comparison alternatives considered). Table 3.3 shows the results of these experiments. For both Little and Big step granularity, we tabulate several items: (1) the total exploration time in seconds, (2) the maximum memory usage in MB, (3) the number of states visited during the entire exploration (note that revisiting the same state multiple times is greatly reduced by the use of state comparison), (4) the maximum exploration depth (i.e., the number of steps in the longest exploration path), and (5) the total number of messages delivered across all exploration paths.

Recall from Section 3.5.3 that when using Little step granularity, a step (i.e., transition) corresponds to the delivery and processing of a single message. However, when using Big step granularity, the delivery and processing of multiple messages are combined into a single step. In the absence of state comparison (which identifies opportunities to prune paths from the state space), using Big steps would result in the same number of messages being delivered but fewer states. Additionally, we would expect some amount of savings compared to Little steps due to less overhead for state management. However, as the results in Table 3.3 show, the use of Big step granularity combined with state comparison has a negative impact on the overall exploration cost. For the subjects we evaluated, Little step granularity results in explorations that are faster and require less memory than explorations using Big step granularity. This is due to increased opportunities for state pruning exposed by using the smaller step size for our subject programs. In other words, when we use Big steps, state comparison is not performed after the processing of every message. As a result, Basset does not detect previously seen states and does not prune the exploration.

Even though combining multiple message deliveries into a single Big step reduces the number of steps for an exploration, the higher level of pruning obtained using Little steps can still reduce the number of steps by an even greater amount than Big steps. This is the case for 4 out the 10 experiments we ran. And when we consider the overall execution time, the number of messages delivered is even more important than the number of states. In all cases, the use of Little steps resulted in both a lower number of messages delivered and an execution time that was less than or equal to that obtained using Big steps.

Given the limited number of subjects used in these experiments, it may be premature to say that Little step granularity is *always* faster. Subjects may exist where Big step granularity could

provide better performance. Whether such subjects exist or if there are other circumstances where combining steps results in faster exploration remain open questions for future investigation.

3.11 Summary

In this chapter we described Basset, a general framework for the systematic testing of Java-based actor programs. We instantiated it for two systems: the Scala programming language and the ActorFoundry library. Our experience with Basset suggests that a general purpose framework for automated testing of actors can be efficient and effective. Experimental results show that using Basset to explore the state space of actor program executions is more efficient than directly exploring the code and its libraries. Experiments also suggest that Basset can effectively explore executions of actor programs, as demonstrated by the discovery of a previously unknown bug in a sample Scala code available from the ScalaWiki web site (the bug was fixed after the authors confirmed our bug report).

We believe that Basset can serve as an excellent research platform for work on testing actor programs and distributed systems. In fact, it is already being used for such efforts [BKSS11,KTL$^+$]. In the future, we expect Basset to be instantiated for more actor systems. We plan to investigate further capabilities and optimizations in Basset, and to use it to test other actor-oriented and message passing-based applications. Basset simplifies the development of tools for the automated testing of programs in new actor languages and runtime libraries, while at the same time making new techniques for testing implemented in the framework readily available for its tool instantiations.

Chapter 4

Dynamic Partial-Order Reduction Heuristics

In this chapter we present our examination of using message-ordering heuristics to improve the efficiency of dynamic partial-order reduction (DPOR) techniques for actor programs.[1] This work was performed using the Basset framework described in Chapter 3. While evaluating Basset, we observed that the order in which messages are considered for exploration at a given state can affect the number of states that can be pruned by a DPOR algorithm. Our results in this chapter show that proper selection of message-ordering heuristics can reduce the number of states explored by over two orders of magnitude.

As discussed in Chapter 1, we considered four questions in our investigation:

- What are some of the natural *heuristics* for ordering scheduling decisions in DPOR for message-passing systems?
- What is the impact of choosing one heuristic over another heuristic?
- Does the impact of these heuristics depend on the DPOR technique?
- Can we predict which heuristic may work better for a particular DPOR technique or subject program?

The remainder of this chapter is organized as follows. In Section 4.1 we present an example which illustrates how the order in which messages are selected for exploration can impact the overall efficiency of DPOR pruning. In Section 4.2 we present a set of heuristics that we have seen being

[1] Some of the material presented in this chapter, as well as Sections 1.4, 2.2, 2.6, and 2.7, is derived from previously published work [LKMA10]. The original publication is available at `www.springerlink.com`, and the copyright is held by Springer-Verlag. It is included with kind permission from Springer Science+Business Media: D.S. Rosenblum and G. Taenzer (Eds.), Fundamental Approaches to Software Engineering 2010, Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques, Volume 6013 of Lecture Notes in Computer Science, 2010, pages 308–322, Steven Lauterburg, Rajesh Karmani, Darko Marinov and Gul Agha (Authors). © Springer-Verlag Berlin Heidelberg 2010.

```
class Master extends Actor {
  ActorName[] workers;
  int counter = 0;
  double result = 0.0;
  public Master(int N) {
    workers = new ActorName[N];
    for (int i = 0; i < N; i++)
      workers[i] =
        create(Worker.class, i, N);
  }
  @message
  void start() {
    int n = 1000;
    for (ActorName w: workers)
      send(w, "intervals", self(), n);
  }
  @message
  void sum(double p) {
    counter++;
    result += p;
    if (counter == workers.length) {
      for (ActorName w: workers)
        send(w, "stop");
      destroy("done");
    }
  }
}
```

```
class Worker extends Actor {
  int id;
  int nbWorkers;
  public Worker(int id, int nb) {
    this.id = id;
    this.nbWorkers = nb;
  }
  @message
  void intervals(ActorName master, int n) {
    double h = 1.0 / n;
    double sum = 0;
    for (int i = id; i <= n; i += nbWorkers) {
      double x = h * (i - 0.5);
      sum += (4.0 / (1.0 + x * x));
    }
    send(master, "sum", h * sum);
  }
  @message
  void stop() { destroy("done"); }
}

class Driver extends Actor {
  static void main(String[] args) {
    ActorName master =
      create(Master.class, args[0]);
    send(master, "start");
  }
}
```

Figure 4.1: ActorFoundry code for the pi example

used or that we felt were reasonable candidates for use. In Section 4.3 we present the results of an evaluation of the heuristics we identified along with our initial insights into why the heuristics performed as they did. And finally, in Section 4.4 we conclude our discussion of this work.

4.1 Illustrative Heuristics Example

To illustrate key DPOR concepts and how different message orderings can affect the exploration of actor programs, we use a simple example actor program that computes the value of π. It is a porting of a publicly available MPI example [Pi], which computes an approximation of π by distributing the task among a set of worker actors.

Figure 4.1 shows a simplified version of this code developed using the ActorFoundry library. The `Driver` actor creates a *master* actor which in turn creates a given number of *worker* actors to

carry out the computation. The `Driver` actor sends a `start` message to the master actor which in turn sends messages to each worker. The worker actors perform their part of the computation and send the result back to the master actor, which collects and reduces the partial results. After all results are received, the master actor instructs the workers to terminate and then terminates itself. All messages in this example are sent asynchronously using ActorFoundry's `send` method (i.e., no RPC-style synchronous calls are used). See Section 2.2 for more details about ActorFoundry.

Figure 4.2 shows the search space for this program with master actor M and two worker actors A and B. Each state in the figure contains a set of messages. A message is denoted as X_Y where X is the actor name and Y uniquely identifies the message to X. We assume that the actors are created in this order: A, B, M. (Note: since the `Worker` actors are created by the `Master` actor's constructor, they are actually created and assigned actor ids *before* the `Master` actor.) Transitions are indicated by arrows labeled with the message that is received, where a transition consists of the delivery of a message up to the next delivery.

The *boxed* states indicate those states that will be visited when the search space is explored using a DPOR technique, and when actors are chosen for exploration according to *the order in which the receiving actors are created*. Namely, the search will favor exploration of messages to be delivered to A over those to be delivered to B or M, so if in some state (say, the point labeled K) messages can be delivered to both A and B, the search will first explore the delivery to A and only after that the delivery to B. To illustrate how this ordering affects how DPOR prunes execution paths, consider the state at point G. For this state, the algorithm will first choose to deliver the message B_1. While exploring the search space that follows from this choice, all subsequent sends to actor B are causally dependent on the receipt of message B_1. This means that DPOR does not need to consider delivering the message M_A before B_1. This allows pruning the two paths that delivering M_A first would require. Similar reasoning shows that DPOR does not need to consider delivering B_2 before A_2 at points S and T, and that it does not need to consider delivering B_1 at point K. In total, this ordering prunes *10 of 12* paths, i.e., with this ordering, only 2 of 12 paths are explored.

The *shaded* states indicate those states that will be visited when the search space is explored using the same DPOR, but when actors are chosen for exploration according to *the reverse-order*

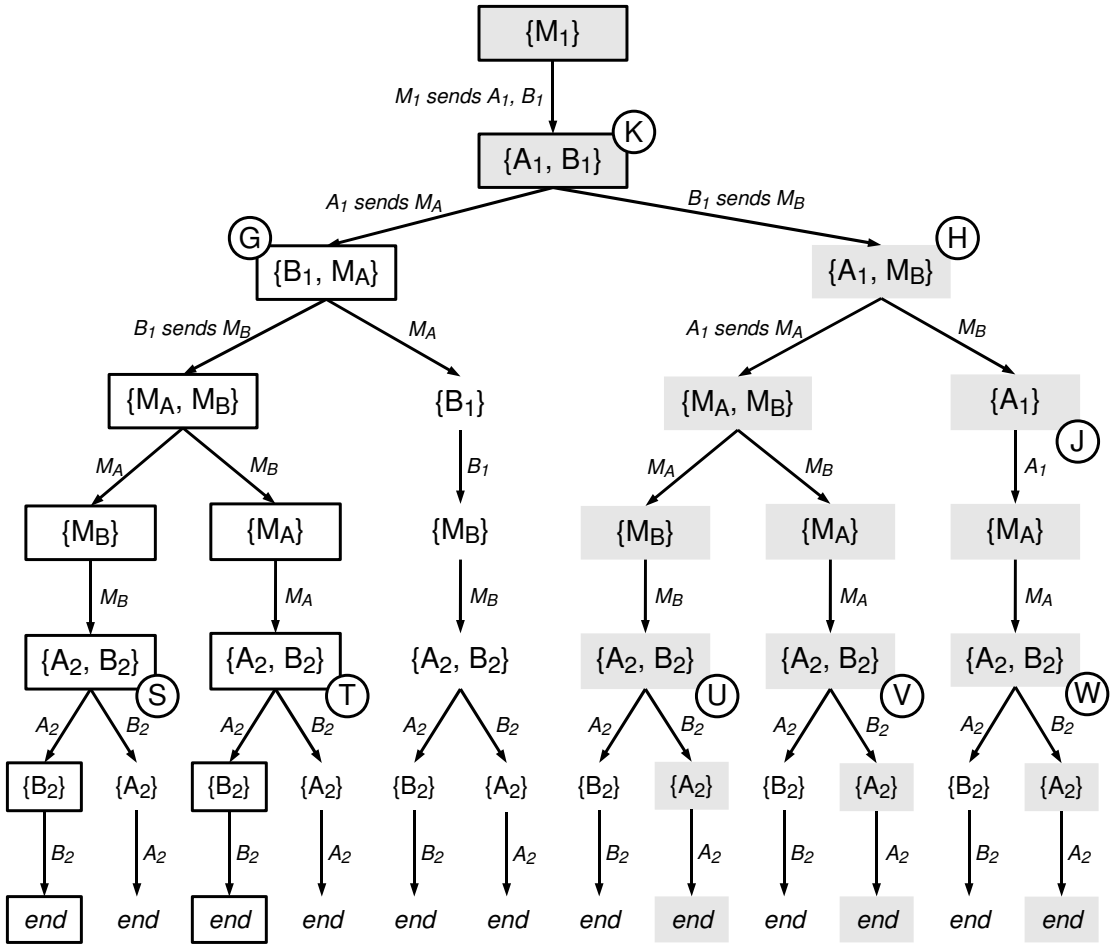

Figure 4.2: State space for pi example with two worker actors

in which the receiving actors are created. This means that the search will favor exploration of messages to be delivered to M over those to be delivered to B or A. This reverse-ordering causes DPOR to prune execution paths differently. Consider the state at point H. For this state, the algorithm will first choose to deliver the message M_B. Following this path, it comes to point J, where the delivery of message A_1 results in message M_A being sent. This send to actor M is *not* causally dependent on the receipt of message M_B. This means that the DPOR also needs to consider delivering the message A_1 before M_B at point H. As the search continues, it discovers that it does not need to consider delivering A_2 before B_2 at points U, V, and W; and also it does not need to consider delivering A_1 at point K. In total, the reverse-ordering prunes *9 of 12* paths,

which is one fewer than when the messages are selected in the order in which the receiving actors are created. As shown in Section 4.3, this difference in the number of paths pruned increases as the number of worker actors increases.

We can see from this simple example that the order in which messages are selected can impact the efficiency of DPOR. In the next section we consider several heuristics for message ordering, and then in Section 4.3 we evaluate the effectiveness of those heuristics.

4.2 Natural Heuristics

The example in Section 4.1 illustrates the idea that scheduling decisions may affect the efficiency of DPOR techniques. In the algorithms presented in Section 2.7, the scheduling choices are represented by the calls to the *choose* method (underlined). The DPOR algorithms implemented in the Basset framework first collect all possible enabled messages for the actors at a given state, and then explore some ordering for processing this set of messages. The key question, therefore, is how do we determine the order in which messages are considered for a given state.

We present eight possible heuristics for ordering messages:

1. *Earliest created actor (ECA)* sorts the enabled actors by their creation time in the *ascending* order. The intuition is to capture the "asymmetry" between some actors in terms of the communication pattern.

2. *Latest created actor (LCA)* is similar to ECA but sorts the enabled actors by their creation time in the *descending* order.

3. *Queue (FIFO)* sorts the actors based on the time of the *earliest* message sent to them, in the *ascending* order. This heuristic captures the common implementation order of choosing messages from a scheduling queue.

4. *Stack (LIFO)* sorts the actors based on the time of the *last* message sent to them, in the *descending* order.

5. *Lowest number of deliverable messages (LDM)* sorts the actors by the number of messages in their respective message queue, in the *ascending* order. The intuition is that the actors that

have received more messages so far in the computation are likely to receive more messages later in the computation.

6. *Highest number of deliverable messages (HDM)* sorts the actors by the number of messages in their respective message queue, in the *descending* order.

7. *Highest average messages sent (HMS)* prioritizes the actors which have been sending the highest number of messages per received message, based on the exploration history. The intuition is that the actors that have been sending more messages in the past are more likely to send more messages in the future.

8. *Send graph reachability (SGR)* is based on information collected during prior executions. Specifically, it maintains a directed graph where nodes represent actors and edges indicate that a message was sent from the first node to the second at some point in the exploration. Now, consider two messages: one to actor A, and one to actor B. If actor B is reachable in the graph from actor A and no such path exists from actor B to actor A, then SGR will prioritize actor A over actor B. The intuition is that actor B is less likely to cause a message to be sent to actor A.

These eight heuristics capture some intuition based on the functioning of the DPOR algorithms and on the patterns of communication in actor programs. While our list of heuristics is not complete by any means, we believe that it is sufficiently representative to help us develop a better understanding of how the use of heuristics can affect the effectiveness of DPOR algorithms.

4.3 Heuristics Evaluation

To evaluate the different heuristics for dynamic partial-order reduction, we conducted experiments using three different DPOR techniques: one based on the algorithm used for dCUTE [SA06] (see Section 2.7.2) and the other two based on dynamically computing persistent sets [FG05, God96] (see Section 2.7.1). The persistent set technique was considered both stand-alone and in combination with sleep sets [God91, God96]. Each of the heuristics and DPOR techniques were implemented in the Basset framework presented in Chapter 3.

		dCUTE		Persistent			dCUTE		Persistent	
Heur.	Subject	# of Paths	time [sec]	# of Paths	time [sec]	Subject	# of Paths	time [sec]	# of Paths	time [sec]
ECA		3821	209	19683	1119		**288**	**18**	**288**	**18**
LCA		**216**	**15**	**216**	**15**		5970	326	5970	326
FIFO		972	60	3240	179		1794	95	1791	94
LIFO	chameneos	2031	108	4899	275	pipesort	1080	54	1080	56
LDM		753	46	3375	184	size=4	384	23	384	24
HDM		3821	215	19683	1206		2072	110	1480	85
HMS		3691	218	19683	1217		307	19	307	19
SGR		3821	204	19683	1169		**288**	**18**	**288**	**18**
ECA		684	49	327	24		7038	411	3822	267
LCA		**16**	**4**	**16**	**4**		**32**	**5**	**32**	**7**
FIFO		68	7	40	5		572	36	368	37
LIFO	fib(5)	81	10	81	10	quicksort	243	20	243	24
LDM		508	37	261	20	size=6	6390	357	2502	152
HDM		526	40	263	22		5118	315	2804	193
HMS		82	9	66	8		195	16	183	16
SGR		684	49	327	25		7038	401	3822	246
ECA		101	7	101	7		516	20	392	16
LCA		188	10	188	10		680	23	640	21
FIFO		122	8	119	7		**360**	**15**	**238**	**10**
LIFO	leader	125	8	125	8	shortpath	859	29	750	25
LDM		133	9	133	9	graph A	585	23	492	19
HDM		**88**	**7**	**88**	**7**		562	23	419	17
HMS		141	9	126	8		540	21	453	17
SGR		101	7	101	7		516	20	392	15
ECA		**120**	**13**	**120**	**13**		7216	239	2658	67
LCA		945	81	19845	1952		7462	366	1865	62
FIFO		**120**	**14**	**120**	**17**		**3488**	**194**	**528**	**28**
LIFO	pi	945	82	19845	2145	shortpath	6472	295	2638	184
LDM	5 workers	**120**	**14**	**120**	**37**	graph B	7326	340	1178	44
HDM		706	97	3424	333		13438	716	2756	134
HMS		945	140	19845	2154		3618	195	783	24
SGR		153	17	567	121		7216	266	2658	68

Table 4.1: Comparison of different ordering heuristics (the best results are in bold)

We first describe the subject programs used to quantitatively evaluate the heuristics. We then present experimental results comparing the different heuristics for two of the DPOR techniques, namely dCUTE and persistent sets without sleep sets. We then present additional results for experiments where we included the use of sleep sets. All experiments are performed using Sun's JVM build 1.6.0_24-b07 on a 2.80GHz Intel Core2 Duo running Ubuntu release 10.04 LTS.

4.3.1 Subject Programs

Our experiments use the eight actor subjects listed in Table 4.1. All of the programs are either originally written using the ActorFoundry library [Agh86, AMST97] or ported to that environment.

The `pi` subject is the example described in Section 4.1. However, the results shown here are for a configuration using five worker actors. Two of the subjects implement more complex algorithms previously used in the dCUTE study [SA06]: `leader` is an implementation of a leader election algorithm; and `shortpath` is an implementation of the Chandy-Misra's shortest path algorithm [CM82]. The `shortpath` subject appears twice in the results: once for a graph with 4 nodes (`shortpathA`), and again for a graph with 5 nodes (`shortpathB`). Note that the two graphs are dissimilar. The `fib` subject computes the n-th element in the Fibonacci sequence. `quicksort` is an implementation of a distributed sorting algorithm that uses a standard divide-and-conquer strategy to carry out the computation. `pipesort` is a modified version of the sorting algorithm used in the dCUTE study [SA06]. `chameneos` is an implementation of the `chameneos-redux` benchmark from the Great Language Shootout (`http://shootout.alioth.debian.org`). Several of these subjects (specifically, `fib`, `pi`, `pipesort`, `quicksort` and `shortpath`) are included as part of the publicly available release of Basset. The software can be downloaded from either the NASA JPF website [JPF] or the Basset homepage [Bas].

4.3.2 Results and Observations

Table 4.1 shows the results of experiments comparing the different heuristics for the DPOR based on persistent sets and the one used for dCUTE. For each heuristic, we tabulate the total number of paths executed and the total exploration time in seconds. The results suggest that the efficiency of the two DPOR techniques is greatly dependent on the order in which messages are selected for exploration.

Recall the four research questions posed at the beginning of this chapter. The first question has been discussed in Section 4.2 where we describe some intuitive ordering heuristics to guide DPOR algorithms. We address the remaining three questions now by making observations on the results

in Table 4.1.

What is the impact of choosing one heuristic over another heuristic?

Table 4.1 shows that for 6 out of 8 experiments, one of the heuristics (but not necessarily the same) performs the best, i.e., there is no tie for the best performing heuristic. In the case of `pipesort` the tie between ECA and SGR is due to the relationship between the two heuristics. Specifically, ECA is the tie-breaking heuristic for SGR.

SGR performs the same as ECA for 6 out of 8 experiments. However, for the remaining two experiments, SGR performs worse than ECA. This suggests that the SGR heuristic, despite its usage of additional information, does not offer any advantage over ECA.

We also observe that the difference between the best and the worst heuristic can be very large. For example, for the `quicksort` subject sorting an array of size 6 and the dCUTE DPOR, the best heuristic (LCA) has *two orders of magnitude* (more precisely, 220X) fewer executions than the worst performing heuristic (ECA). Note that both these heuristics are natural orders on the scheduling queue. In fact, the dCUTE DPOR algorithm as originally presented [SA06] employs the ECA ordering. The second best performing heuristic (HMS) for `quicksort` still explores 6 times as many executions as the best heuristic. For the other subjects, the ratio between the number of executions in the worst and the best case ranges from 2X (for `leader`) to 91X (for `chameneos`).

In general, the exploration time strongly correlates with the number of executed paths. This observation suggests that the better heuristics do not have a significant computation cost, and thus their reduction in the number of executions directly translates into savings in the exploration time. There are exceptions: for the subject `shortpathB`, the exploration time does not correlate with the number of paths executed as closely as other experiments. We believe that this is due to our experiments using Basset which is built on top of JPF and uses checkpointing and restoring to explore different paths, rather than re-execution. Hence, the time may relate more to the number of states visited instead of the number of executions, or stated differently, the time may depend more strongly on the length of all executions instead of the number of executions.

Does the impact of these heuristics depend on the DPOR technique?

Although the results differ between the two DPOR algorithms for the experiments, the results

exhibit a *similar ranking* of heuristics for both algorithms. In other words, for a given subject, heuristics that perform well for one DPOR technique tend to perform well for the other. Similarly, a heuristic that performs poorly typically does so for both DPOR algorithms.

It is evident from the table that for all 8 experiments, the best heuristic exactly matches for both DPOR algorithms. Moreover, even the worst heuristic matches for 7 out of 8 experiments.

Can we predict which heuristic may work better for a particular DPOR technique or subject program?

We found that which heuristic performs the best relates to the communication patterns employed by the program. For example, in a *pipelined computation* (e.g., `pipesort`), it is more efficient to schedule first the actors that represent the early stages in the pipeline. On the other hand, in a *divide-and-conquer tree* (e.g., `fib`), it is more efficient to schedule child actors before the parent actor.

Indeed, the ECA heuristic is the best performing heuristic for `pipesort`. ECA prioritizes actors in the early stages of a pipeline, and this enables the DPOR algorithms to collect all possible messages for actors in the later stages of the pipeline.

For 3 out of 8 subjects, the LCA heuristic performs the best among all heuristics. Two of these subjects—`fib` and `quicksort`—employ a divide-and-conquer approach. The remaining subject, `chameneos`, has a request-reply pattern between a broker and many clients. LCA allows the DPOR algorithm to collect all possible messages sent from the clients to the broker before exploring all the permutations of this set of messages.

For subjects with arbitrary graphs and communication patterns, the FIFO heuristic outperforms the remaining heuristics. For instance, the input graphs for `shortpathA` and `shortpathB` are dissimilar, and the effectiveness of several heuristics varied between the two experiments. Yet, the FIFO heuristic is the most effective heuristics for both inputs.

We performed some additional experiments for `shortpath` (not shown in the table) to identify how much the choice of heuristic depends on the program *input* rather than program *code*. In particular, the input to `shortpath` is a graph, and the messages exchanged depend on the topology of this graph. We considered seven more graphs (all with four or five nodes) in addition to the two

for which the results are shown. While there is some variation of the results, in all the cases, FIFO is the best heuristic, either by itself, or together with some other heuristics (e.g., for a graph that is a list, there is only one execution path for any heuristic). These results are not conclusive, but they strongly suggest that the choice of heuristic depends on the program (and its communication pattern) more than on the input. We believe that it would be worthwhile to evaluate how `shortpath` performs for all graphs of various given sizes. These and other more exhaustive experiments using different communication patterns may be able to shed more light on how communication patterns and topology affect heuristic performance and could lead to better heuristic selection. We leave such extended studies for future work.

In summary, the results suggest the following set of guidelines for selecting a heuristic before the exploration of a program. (1) If there is no well-defined topology and communication pattern in the program (or if this communication pattern is not known a priori), then the default heuristic should be FIFO, since it is never the worst and sometimes is even the best heuristic. (2) If the communication pattern is a pipeline, then ECA should be used. (3) If the communication pattern is a divide-and-conquer tree, then LCA should be used.

4.3.3 Heuristics and Sleep Sets

In the previous subsection we evaluated several heuristics using two different but similar dynamic partial-order reduction implementations. To see how heuristics would perform under more disparate conditions, we performed a series of experiments using persistent sets augmented with *sleep sets*. We implemented a variant of the persistent set algorithm which includes sleep sets in our Basset framework. Sleep sets is a partial-order reduction technique, which is based on the history of exploration [God91]. Specifically, sleep sets record the transitions that have already been explored from a particular configuration, and avoid exploring them in successor configurations until some condition is met. Sleep sets can further prune the number of transitions and paths that are explored over using persistent sets alone [God96].

In Table 4.2 we present the results of experiments comparing the different heuristics for persistent sets both with and without the addition of sleep sets. As in Table 4.1, we tabulate the total number of paths executed in their entirety and the total exploration time in seconds. We

		Persistent			Persistent+Sleep				Persistent			Persistent+Sleep		
Heur.	Subject	# of Paths	# of Trans.	time [sec]	# of Paths	# of Trans.	time [sec]	Subject	# of Paths	# of Trans.	time [sec]	# of Paths	# of Trans.	time [sec]
ECA		19683	118198	1119	216	1827	21		288	1294	18	288	1294	18
LCA		216	1232	15	216	1232	15		5970	32386	326	288	1945	24
FIFO		3240	19460	179	216	1682	20		1791	8563	94	288	1449	19
LIFO	chameneos	4899	27068	275	216	1483	18	pipesort	1080	4762	56	288	1594	20
LDM		3375	18832	184	216	1675	21	size=4	384	1738	24	288	1330	19
HDM		19683	118198	1206	216	1827	23		1480	7246	85	288	1448	21
HMS		19683	117920	1217	216	1832	23		307	1380	19	288	1300	18
SGR		19683	118198	1169	216	1827	24		288	1294	18	288	1294	19
ECA		327	1651	24	16	140	5		3822	16767	267	32	273	8
LCA		16	92	4	16	92	4		32	157	7	32	157	5
FIFO		40	204	5	16	102	4		368	1587	37	32	180	6
LIFO	fib(5)	81	460	10	16	153	5	quicksort	243	1082	24	32	244	8
LDM		261	1290	20	16	143	5	size=6	2502	10396	152	32	276	8
HDM		263	1344	22	16	140	5		2804	12422	193	32	273	8
HMS		66	361	8	16	130	5		183	853	16	32	194	6
SGR		327	1651	25	16	140	5		3822	16767	246	32	273	8
ECA		101	499	7	24	145	4		392	1465	16	126	490	7
LCA		188	917	10	24	159	4		640	2159	21	126	523	7
FIFO		119	576	7	24	148	4		238	911	10	126	474	7
LIFO	leader	125	618	8	24	152	4	shortpath	750	2632	25	126	598	8
LDM		133	661	9	24	157	4	graph A	492	1769	19	126	511	8
HDM		88	442	7	24	145	4		419	1615	17	126	557	8
HMS		126	624	8	24	154	4		453	1596	17	126	512	8
SGR		101	499	7	24	145	4		392	1465	15	126	490	7
ECA		120	932	13	120	932	19		2658	8477	67	296	1032	12
LCA		19845	156071	1952	120	1237	19		1865	7077	62	296	1229	14
FIFO		120	932	17	120	932	14		528	2444	28	296	1409	15
LIFO	pi	19845	156071	2145	120	1237	17	shortpath	2638	10246	184	296	1570	19
LDM	5 workers	120	932	37	120	932	14	graph B	1178	3765	44	296	1051	15
HDM		3424	26800	333	120	1061	16		2756	14714	134	296	1956	30
HMS		19845	156071	2154	120	1237	17		783	2494	24	296	1014	20
SGR		567	4403	121	120	1026	14		2658	8477	68	296	1032	25

Table 4.2: Effect of heuristics on persistent sets combined with sleep sets

also include the total number of transitions executed. We have repeated the results obtained for persistent sets (from Table 4.1) to help put the sleep set results in perspective. The results using sleep sets continue to show that the efficiency of the two DPOR techniques is dependent on the order in which messages are selected for exploration. However, the savings that can be obtained by choosing one heuristic over another is not as pronounced as was seen with our experiments using persistent sets on their own or those using the dCUTE algorithm.

As expected, using persistent sets together with sleep sets often results in a significant reduction in states explored compared to persistent sets on their own. In fact, *all programs for all heuristics have exactly the same number of paths.* To see the savings that results from using different message-ordering heuristics, one needs to look beyond the number of paths executed in their entirety and consider the number of transitions that are executed to explore those paths. A lower number of transitions implies that some of the redundant paths that were ultimately pruned were pruned *earlier* than they were using a different heuristic.

Due to the much better pruning afforded by sleep sets, we see more modest differences in the savings provided by the different heuristics. For example, our persistent set experiments showed that using the ECA heuristic for the `quicksort` subject resulted in 119x more execution paths and 107x more transitions than the LCA heuristic. Using persistent sets combined with sleep sets, however, the number of completed execution paths was the same, and the difference in transitions was less than double (1.74x to be precise). The greatest difference we saw was in our experiments for was for `shortpathB`: the HDM heuristic executed 1.93x more transitions than the HMS heuristic.

For seven of the eight subjects, the best performing heuristic remained the same with or without sleep sets. However, for `shortpathB`, FIFO is replaced by HMS as the best performer. It should be noted, however, that `shortpathB` is also one of the three subjects where adding sleep sets reduced the number of completed execution paths from what was obtained using persistent sets alone (`shortpathA` and `leader` are the other two). In most cases, the number of paths executed using the best heuristic for persistent sets is the same as the number of paths executed by all heuristics using persistent plus sleep sets. In the case of `shortpathB`, the lowest number of paths for persistent set experiments is 528, but the number of paths obtained for all heuristics when using persistent plus sleep sets is 296. This may indicate that a better result for `shortpathB` could be obtained by

persistent sets using a message-ordering scheme other than the eight heuristics we identified.

Although the use of different heuristics with sleep sets did not provide savings as significant as those seen for the dCUTE algorithm or for persistent sets alone, our experiments show their use is still clearly beneficial. And in cases where sleep sets are not in use, the benefit of considering message ordering is quite large – over two orders of magnitude in some cases. One should also consider that the overhead of using sleep sets in conjunction with persistent sets is greater than that for using persistent sets alone. The experiments that we performed were done in an environment for systematically testing actual code. In such an environment, the cost of executing transitions is much higher than it would be if were model checking a simpler model. In the latter case, the overhead of using the sleep sets extension could have a more detrimental impact on the overall performance of the model checking exercise, thus contraindicating its use.

4.4 Summary

Based on our work with actor systems, we believe that systematic exploration of message schedules is a viable approach to address the important but challenging problem of testing actor programs. As shown in this chapter, dynamic partial-order reduction (DPOR) techniques can significantly speed up systematic exploration, but they are also highly sensitive to the order in which messages are explored. We described and compared several heuristics that can be used for ordering messages. Our results show up to two orders of magnitude difference in the number of executions explored based on which heuristic is used. Moreover, our analysis of the results identified some initial guidelines that, based on the type of program, can aid selection of a good message-ordering heuristic before the exploration. Questions remain, however. For instance, how can one balance the potentially conflicting goals of improving DPOR performance and guiding exploration for the purpose of more quickly finding bugs? Given the growing use of systematic testing as an approach to improving the quality of actor-based and distributed systems, further investigation into message-ordering heuristics should be considered.

Chapter 5

Related Work

This chapter provides a brief overview of work related to this dissertation. The work most related to the Basset framework is on model checking actor programs. Sen and Agha present dCUTE [SA06] which checks actor programs written using a simplified actor library, by re-executing the programs for various message schedules. Both dCUTE and Basset use a dynamic partial-order reduction based on the happens-before relation to avoid exploring equivalent schedules. dCUTE combines this partial-order reduction with a mixed concrete and symbolic execution for test generation [GKS05, SMA05, CDE08]. In contrast, Basset provides a common framework for *stateful* exploration of Java-based actor libraries and handles *full actor libraries* for Scala and ActorFoundry (including dynamic creation and destruction of actors). Also, Basset is built on top of JPF and can reuse its functionality for state-space exploration (e.g., heuristics for ordering exploration).

The Erlang language has also received a lot of attention. Fredlund and Svensson present McErlang [FS07], a stateful model checker for actor programs written in the Erlang programming language [Arm07]. McErlang, which is itself written in Erlang, modifies the concurrency system of the Erlang run-time library. A previous model checker for Erlang, etomcrl [AE01], checked Erlang programs by translating them into μCRL [GP95] and using off-the-shelf model checkers. This approach is similar to the very first version of JPF [Hav99] which checked Java programs by translating them into Promela and using SPIN [Hol97].

Another approach by Claessen et al. [CPS+09] proposes finding race conditions in Erlang code during unit testing by using Quviq QuickCheck [Hug07] and a new randomizing scheduler for Erlang called PULSE. This scheduler chooses and records the order in which messages are delivered during testing. However, since it randomly chooses processes to run, it does not facilitate the systematic exploration of all possible message schedules like Basset. Furthermore, Basset does not focus on one language/library but provides a general framework built on an existing tool (JPF)

and additionally incorporates several existing optimizations for exploration.

Bordini et al. [BFVW06] present a translation from AgentSpeak, a widely used agent-oriented programming language, into Java bytecode so that the original program could be verified using JPF. Agents in AgentSpeak share many similarities with actors. For instance, an agent communicates only by exchanging messages, and it has a private mailbox to queue up messages that cannot be processed immediately, just like actors. While their work focused on mapping AgentSpeak features into Java constructs, so that the resulting program can be executed in JPF, we instead focused on creating an extension for JPF so that different Java-based actor systems can be tailored to Basset with a minimum effort.

Also related to Basset is work on checking distributed systems [AG06, BB07, Sto00, HGC04, YCW+09, YKKK09]. In particular, Artho and Garoche [AG06] and Barlas and Bultan [BB07] provide frameworks for executing distributed Java code in JPF. A key problem is that such code uses network calls that JPF does not support as they depend on native code from the Java standard libraries. Artho and Garoche solve this problem by instrumenting the bytecode, whereas Barlas and Bultan's NetStub framework utilizes stub classes [BB07]. These solutions are conceptually similar to Basset in that they replace/avoid the standard Java network library similar to how Basset replaces actor libraries. These solutions would be also valuable for checking migration of actors. However, both frameworks focus on low-level communication, whereas Basset focuses on high-level exploration of possible behaviors for actor programs.

Also along these lines, Yang et al. developed MODIST [YCW+09] to model check unmodified distributed systems (that may or may not be actor-based) in a Windows environment. Their approach inserts a thin layer between the application processes and the operating system to intercept and systematically explore system actions with an independent model checking engine. MODIST also simulates potential sources of errors such as system crashes and message reordering. In contrast to work that focuses on testing systems before moving them into production, Yabandeh et al. propose an approach to identifying and avoiding errors in deployed distributed systems written in the Mace language [KAB+07]. Their CrystalBall [YKKK09] approach continuously performs state-space exploration of a system's global state and its potential behavior. This is done concurrently with the actual execution of the production system. Rather than attempting to exhaustively

explore a systems behavior from an initial state (like Basset), the approach explores behavior based on current system states and attempts to steer system execution away from potential errors.

Basset was developed to support the systematic testing of actual program code. Although not specifically targeted at actor-based or distributed systems, several other tools which systematically explore actual code for concurrent systems have been introduced over the years. VeriSoft [God97], the first model checker of this sort, targets code written in the C language. CMC [MPC+02] also targets C and additionally C++. Tools such as BogorVM [RDH03], Java PathFinder [VHB+03] and JNuke [ASB+04] are focused on software written in the Java language. CHESS [MQ07] is targeted at Microsoft's .NET and also supports the Win32 and Singularity platforms.

Partial-order reduction is an important optimization for alleviating the state-space explosion in model checking [CGP99, ABH+97, FG05, God96, YCW+09]. As discussed in Section 3.5.2, Basset provides dynamic partial-order reduction capabilities based on the happens-before relation in order to avoid exploring unnecessary message schedules. It also facilitates state-space reduction through the use of state comparison to determine when to prune exploration. At present, the use of dynamic partial-order reduction and the use of state comparison are mutually exclusive in Basset. Recent work by Yang et al. [YCGK08] and Yi et al. [YWY06] proposes combining these two optimizations, which we plan to explore as a possible capability for Basset in the future.

Chapter 6

Conclusions

This dissertation presents our efforts to help alleviate some of the difficulties that arise when testing actor programs. For example, one goal of our work was to address the inefficiencies that arise when testing actor systems that include complex, multithreaded runtime architectures. The complexity of an actor system's underlying architecture interferes with our ability to test and explore what is really important: the functionality and interaction of the actors themselves. If we consider the goal of checking actor systems to be the testing of the actor programs themselves and not the underlying library used to implement them, we can define more tractable testing scenarios. This does not mean that the underlying runtime architectures do not need to be tested. They do. Rather, we view the Basset framework's approach of testing just the actors themselves as an application of the principle of *separation of concerns* to testing.

Another goal of our work was to provide a means to leverage similarities across different actor systems and to demonstrate that it is possible to build reusable components and techniques. To address these goals, we propose a general framework and environment for the systematic testing of actor programs that compile to Java bytecode. In brief, our thesis is the following:

> *It is possible to build a general framework that (1) allows efficient exploration of actor programs written in languages that compile to Java bytecode and (2) facilitates the reuse of testing capabilities across such languages.*

We described Basset, a general framework and systematic testing environment for exploring Java-based actor programs. We implemented the Basset framework on top of Java PathFinder (JPF), and created tool instantiations of the framework for two systems: the Scala programming language and the ActorFoundry library. Our experience with Basset shows that a general purpose framework for automated testing of actors can be efficient and effective. Our experimental results

show that using our Basset-based tools to explore the state space of actor program executions is more efficient than directly exploring the code and its libraries. For instance, a simple `helloworld` actor program and its simplified runtime library took over 7 minutes to explore using JPF. The same actor program could be explored in less than a second using Basset.

We demonstrated Basset's ability to support the reuse of exploration capabilities by developing two state-space reduction techniques. The first, a specialized state comparator for the stateful search of actor programs enables more aggressive pruning of the search space than the generic comparator provided with JPF. The second technique is dynamic partial-order reduction (DPOR) for actor programs. We implemented multiple DPOR algorithms in the core of the Basset framework, and each is available for use by both the ActorFoundry and Scala tool instantiations. While implementing these DPOR algorithms, we noted that although they significantly sped up systematic exploration, they were highly sensitive to the order in which messages are explored. This observation led to our investigation of the impact that message-ordering heuristics can have on reducing the exploration state space.

We described and compared several heuristics that can be used for ordering messages. Our results show up to two orders of magnitude difference in the number of executions explored can be realized based on how messages are ordered. Moreover, our analysis of the results discovered guidelines that, based on the type of program, can aid selection of a good heuristic before the exploration.

We next reiterate the contributions of this dissertation and conclude with some final remarks.

6.1 Contributions

This dissertation makes the following contributions:

- **I**ntroduces the concept of a general framework for exploration of actor programs that explicitly takes into account the nature of these programs.

- **P**rovides an implementation of the general framework concept in a tool called Basset which uses the Java PathFinder model checker. Basset has been released as a publicly available extension for JPF called jpf-actor. Basset can be downloaded from either the NASA JPF

website [JPF] or the Basset homepage [Bas].

- **P**rovides instantiations of the Basset framework for actor programs written in the ActorFoundry library and in the Scala programming language. These instantiations illustrate the viability of and value provided by the framework concept in general and the Basset framework in particular. The instantiations also provide the first state exploration engines for these two actor systems, which are based on very different design decisions.

- **I**ncorporates two known optimization techniques: dynamic partial-order reduction [SA06, CGP99] and state comparison/hashing [SL08, CGP99] in Basset. Due to the nature of the Basset framework, these two techniques for speeding up exploration were automatically available for use with both the ActorFoundry and the Scala instantiations of the Basset framework, thus illustrating the reuse of tools and techniques made possible by the common framework concept.

- **E**valuates the Basset framework on several subjects. The evaluation shows that a single framework can systematically explore programs in an effective manner for multiple languages. Additionally, we show that Basset's approach of exploring only application code (and not underlying system libraries) allows for more efficient exploration than if the underlying runtime architecture of the actor system were also considered.

- **I**dentifies and presents eight ordering heuristics that can be applied when using dynamic partial-order reduction to limit the search space during systematic testing of actor-based programs.

- **E**valuates these ordering heuristics for three DPOR techniques: one based on the algorithm used for dCUTE and two others based on persistent sets. The persistent set technique was considered both stand-alone and augmented with sleep sets.

- **S**ummarizes the observed advantages and disadvantages of the identified heuristics, and presents preliminary guidelines regarding the use of heuristics based on the characteristics of the program under test.

6.2 Final Remarks

We believe that our work significantly enhances the viability of systematic testing and message schedule exploration as an approach to addressing the important but challenging problem of testing actor programs. At the same time, we realize that there are many related opportunities to build on this work.

The Basset framework has been designed and implemented with extensibility in mind and is intended to serve as a platform for further research into the testing of actor programs. We expect that this research will directly support and facilitate additional work in this area. In fact, Bokor et al. have already used our Basset framework for their work on model checking fault-tolerant distributed protocols [BKSS11]. Basset has also been used as the development platform for work by Karmani et al. on the development of improved dynamic partial-order reduction techniques for actor programs [KTL$^+$] (this work has recently been submitted for publication). We also feel that there is additional work in the area of DPOR heuristics. For instance, there has been recent work on combining DPOR techniques with stateful exploration [YWY06, YCGK08], and we plan to evaluate the effectiveness of heuristics for such approaches.

Basset simplifies the development of environments for the testing of programs in new Java-based actor languages and runtime libraries. As a result, we expect to instantiate the framework for more actor systems. For instance, preliminary work has already been performed for a Basset-based tool for Erjang [Erj], a Java-based virtual machine for the Erlang programming language [Arm07].

Parallel and distributed programming are becoming the norm, and message passing-based approaches such as the actor model offer a promising alternative for developing parallel and distributed code. Testing such code, however, is extremely challenging due to the non-determinism of message delivery schedules. Nonetheless, as evidenced by the growing number of actor-oriented languages and libraries, the actor model is increasing in popularity. In this dissertation, we presented Basset, a framework to support the systematic testing of actor-based programs. Basset supports the development of testing tools for Java-based actor languages and libraries. Perhaps equally important, Basset offers an environment to explore new ideas for the testing and state-space exploration of actor-based programs and other message-passing systems. It is already being used for this purpose, and we hope that it will continue to be used going forward.

References

[ABH+97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 340–351, London, UK, 1997. Springer-Verlag.

[AE01] Thomas Arts and Clara Benac Earle. Development of a verified Erlang program for resource locking. In *Formal Methods in Industrial Critical Systems*, 2001.

[AF] ActorFoundry webpage. `http://osl.cs.uiuc.edu/af/`.

[AG06] Cyrille Artho and Pierre-Loïc Garoche. Accurate centralization for applying model checking on networked applications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 177–188. IEEE Comp. Society, 2006.

[Agh86] Gul Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[Akk] Akka webpage. `http://akka.io/`.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[AQR+04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV '04*, volume 3114 of *LNCS*, pages 484–487. Springer, 2004.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[ASB+04] Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. JNuke: Efficient dynamic analysis for Java. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV '04*, volume 3114 of *LNCS*, pages 462–465. Springer, 2004.

[Bas] Basset webpage. `http://mir.cs.illinois.edu/basset/`.

[BB07] Elliot Barlas and Tevfik Bultan. NetStub: A framework for verification of distributed Java applications. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 24–33. ACM, 2007.

[BFVW06] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2), March 2006.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[BKSS11] Peter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. Efficient model checking of fault-tolerant distributed protocols. In *Proceedings of the 41st IEEE/IFIP International Confenference on Dependable Systems and Networks*, DSN/DCCS '11, 2011.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CM82] K. Mani Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–837, 1982.

[CPS+09] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 149–160, New York, NY, USA, 2009. ACM.

[d'A07] Marcelo d'Amorim. *Efficient Explicit-State Model Checking of Programs with Dynamically-Allocated Data Structures*. Ph.D., University of Illinois at Urbana-Champaign, October 2007.

[DIS99] Claudio DeMartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[DPE06] Matthew B. Dwyer, Suzette Person, and Sebastian G. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '06, pages 92–104. ACM, 2006.

[EJL+03] Johan Eker, Jrn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - The Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[Erj] Erjang webpage. `https://github.com/trifork/erjang/wiki`.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121. ACM, 2005.

[Fid88] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, pages 183–194, New York, NY, USA, 1988. ACM.

[FS07] Lars-Åke Fredlund and Hans Svensson. McErlang: A model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 125–136, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90, pages 176–185, London, UK, 1991. Springer-Verlag.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

[HA92] Christopher R. Houck and Gul Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume 2 of *ICPP ' 92*, pages 158–165, 1992.

[Hav99] Klaus Havelund. Java PathFinder, A translator from Java to Promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 152–, London, UK, 1999. Springer-Verlag.

[HGC04] Daniel Hughes, Phil Greenwood, and Geoff Coulson. A framework for testing distributed systems. In *Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, P2P '04, pages 262–263, Washington, DC, USA, 2004. IEEE Computer Society.

[HL07] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Operating System Review*, 41(2):37–49, 2007.

[HO07] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION '07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, May 1997.

[Hug07] John Hughes. QuickCheck testing for fun and profit. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages, PADL '07*, volume 4354 of *LNCS*, pages 1–32. Springer, 2007.

[Ios01] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 254–, Washington, DC, USA, 2001. IEEE Computer Society.

[Jet] Jetlang webpage. `http://code.google.com/p/jetlang/`.

[JPF] Java PathFinder webpage. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[Jsa] Jsasb webpage. `https://jsasb.dev.java.net/`.

[JSM+09] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. Symmetry and partial order reduction techniques in model checking Rebeca. *Acta Informatica*, 2009.

[KAB+07] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[Kim97] WooYoung Kim. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. Ph.D., University of Illinois at Urbana-Champaign, May 1997.

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[KSA09] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: A comparative analysis. In *Proceedings of the 7th International Conference on the Principles and Practice of Programming in Java*, 2009.

[KTL+] Rajesh Karmani, Samira Tasharofi, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. A novel partial-order reduction technique for systematically testing actor programs. Submitted for publication.

[KWG09] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Computer Aided Verification, 21st International Conference (CAV)*, volume 5643 of *LNCS*, pages 398–413. Springer, 2009.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LDMA09] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of Java-based actor programs. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society.

[Lea00] Doug Lea. A Java fork/join framework. In *Java Grande*, 2000.

[LKMA10] Steven Lauterburg, Rajesh Karmani, Darko Marinov, and Gul Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In David Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer Berlin / Heidelberg, 2010.

[MD05] Madanlal Musuvathi and David L. Dill. An incremental heap canonicalization algorithm. In *Proceedings of the 12th SPIN Workshop on Model Checking Software (SPIN)*, volume 3639 of *Lecture Notes in Computer Science*, pages 28–42. Springer, 2005.

[Mica] Microsoft. Asynchronous agents library. `http://msdn.microsoft.com/en-us/library/dd492627(VS.100).aspx`.

[Micb] Microsoft. Axum programming language. `http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx`.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D., Johns Hopkins University, May 2006.

[MPC+02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 75–88, New York, NY, USA, 2002. ACM.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[MT97] Ian A. Mason and Carolyn L. Talcott. A semantically sound actor translation. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, ICALP '97, pages 369–378. Springer-Verlag, 1997.

[New] Newspeak webpage. `http://newspeaklanguage.org/`.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[Pi] Pi original source code webpage. `http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/simplempi/main.htm`.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

[Rev] Revactor webpage. `http://revactor.org/`.

[SA06] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, volume 3922 of *LNCS*, pages 339–356. Springer, 2006.

[Sca] Scala webpage. `http://www.scala-lang.org/`.

[ScW] ScalaWiki website. `http://scala.sygneca.com/`.

[SL08] Corinna Spermann and Michael Leuschel. ProB gets Nauty: Effective symmetry reduction for B and Z models. In *Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 15–22, Washington, DC, USA, 2008. IEEE Computer Society.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 104–128. Springer-Verlag, 2008.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[Sto00] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*, 36:20–34, December 2001.

[VCST05] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software engineering*, ESEC/FSE-13, pages 273–282, New York, NY, USA, 2005. ACM.

[Ven09] Bill Venners. Twitter on Scala. *Scalazine*, April 3, 2009. `http://www.artima.com/scalazine/articles/twitter_on_scala.html`.

[VGK08] Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *Computer Aided Verification, 20th International Conference, CAV 2008*, volume 5123 of *LNCS*, pages 66–79. Springer, 2008.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[YCGK07] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Model Checking Software, 14th International SPIN Workshop*, volume 4595 of *LNCS*, pages 58–75, 2007.

[YCGK08] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Model Checking Software, 15th International SPIN Workshop*, volume 5156 of *LNCS*, pages 288–305. Springer, 2008.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Haoxiang Lin Xuezheng Liu, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI '09, pages 213–228. USENIX Association, 2009.

[YKKK09] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, pages 229–244. USENIX Association, 2009.

[YWY06] Xiaodong Yi, Ji Wang, and Xuejun Yang. Stateful dynamic partial-order reduction. In *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *LNCS*, pages 149–167. Springer, 2006.